REMARKS

The application contains claims 1-7, 9-28. Claim 8 has been canceled. All claims stand rejected over 35 U.S.C. § 112, second paragraph, for use of the term 'cachelet.' The specification is objected to for the same reason. Claims 1-7 and 19 stand rejected as anticipated. In view of the foregoing amendments and following remarks, Applicants respectfully request allowance of the application.

THE CLAIMS ARE DEFINITE.

Applicant requests reconsideration of the § 112, second paragraph rejection because the term 'cachelet' is definite, particularly when considered in light of the specification. "***Cachelets***" refers to cache subsystems, essentially mini-caches, that are provided in a common level of cache. The specification provides clear examples of cachelets throughout the discussion, particularly with regard to FIGS. 1 and 4-6.

The suffix "-let" is commonly used in English as a diminutive (Example: piglet, ringlet). This usage recognized by modern dictionaries. See, Webster's Encyc. Unabridged Dictionary, p. 1104 (attached). Given the discussion provided by the specification, there is nothing indefinite or vague about its use to modify the term "cache." Applicants, therefore, respectfully request withdrawal of this ground of rejection.

THE SPECIFICATION PROVIDES ANTECEDENT BASIS FOR 'CACHELETS.'

Applicant requests reconsideration of the objections made in paragraph 3 of the Office Action. The specification provides a clear and comprehensive description of a cachelet as that term is used in the claims. Specifically, FIGS. 1 and 4-6 and their associated discussion provide a description of a cachelet, its use with other components of processing systems and possible structures and operation. The specification provides several examples of cachelets in various operational scenarios. Applicants respectfully suggest that the specification's disclosure provides ample antecedent basis for the claimed subject matter.

THE CLAIMS DEFINE OVER THE CITED ART.

The status of the outstanding rejections is a bit unclear from the description of the Office Action. While the Office Action formally states that claims 1-7 and 19 are rejected over prior art, the detailed analysis includes claims 8-10 as well. For completeness, all of these claims are discussed herein.

Claims 1-7 Define Over The Art.

Claims 1-7 stand rejected over Ayukawa, U.S. Patent No. 6,381,671. Claim 1 has been amended to recite:

> A cache, comprising a plurality of independently addressable cachelets, each cachelet to provide data responsive to load requests in a single clock cycle.

Ayukawa does not teach this subject matter. In Ayukawa's system, there are four DRAM macro structures 5Ma-5Md but only two of them can provide data responsive to a read request in a single clock cycle. The other two DRAM macro structures appear to support write operations. See, Ayukawa, Cols. 17:60-18:6 (two of the DRAM macro structures connected to read buffers 454R, 455R; two connected to write buffers 454W, 455W). Therefore, claim 1 defines over the cited art.

Dependent claims 6 and 7 recite other features that distinguish Ayukawa, including an instruction decoder and load units. The Office Action alleges that the instruction decoder corresponds to a row/column decoder and the load units correspond to busses. Applicants respectfully disagree. These terms refer to well-known components of modern processing systems. An instruction decoder decodes program instructions. A load unit executes load instructions by retrieving data as specified by a program instruction. See, for example, Pentium®Pro Processor System Architecture, Chapter 5, 1999 (attached) (describing instruction decoding and load execution). Applicants respectfully submit that the Office Action's analysis is inconsistent with these well-known principles of computing systems. Therefore, the rejections to claims 6 and 7 should be withdrawn.

__Claims 9-10 Define Over The Art.__

Claim 10 has been rewritten as an independent claim and defines over __Ayukawa__. Claim 10 recites, inter alia:

> if a conflict occurs among cachelet pointers, forwarding one of the data requests associated with a conflicting cachelet pointer to the identified cachelet, and
> reassigning data requests associated with remaining conflicting cachelet pointers to unused cachelets.

__Ayukawa__ does not disclose this subject matter. Instead, he discloses that if two requests are directed to the same DRAM macro structure, only a 'higher priority' request can proceed. The lower priority request must wait until the higher priority request completes. __Ayukawa__, Col. 18:13-30. There is no teaching or suggestion that a conflicting request be forwarded to an unused cachelet as recited in claim 10. Claim 10, therefore, defines over the art.

Claim 9 has been rewritten to depend from claim 10 and also is allowable.

__Claim 19 Defines Over The Art.__

Claim 19 stands rejected as anticipated by __Ayukawa__. Claim 19 has been amended to recite:

> means for distributing independent loads to each of the cachelets in a single clock cycle.

As explained above, __Ayukawa__ does not teach or suggest this subject matter. In his system, there are four DRAM macro structures but only two of them can receive read requests at a time. The other two support write requests. Therefore, claim 19 defines over the art.

__NEW CLAIMS__

New claims 22-28 are presented for examination. Independent claims 22, 24, and 26 correspond to claims 12-13 and 20 as originally presented. Original claims 12-13 and 20-21 were not rejected over prior art in the Office Action.

CONCLUSION

All rejections have been overcome. Applicants respectfully request allowance of the application.

Respectfully submitted,

Date: 6/25/03

[illegible]

Robert L. Hails, Jr.
Registration No. 39,702
(Attorneys for Intel Corporation

KENYON & KENYON
1500 K Street, N.W.
Washington, D.C. 20005
Ph.: (202) 220-4200
Fax.: (202) 220-4201

Pentium® Pro Processor System Architecture

MINDSHARE, INC.

Tom Shanley

ADDISON-WESLEY DEVELOPERS PRESS

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California, • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-47953-2

Sponsoring Editor: Kathleen Tibbetts
Project Manager: Sarah Weaver
Cover Design: Barbara T. Atkinson
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9-MA-0099989796
First Printing, December 1996

Chapter 4: Processor Startup

Chapter 5: The Fetch, Decode, Execute Engine

5 The Fetch, Decode, Execute Engine

The Previous Chapter

The previous chapter described the processor state at startup time. It also described the process that the processors engage in to select the processor that will fetch and execute the power-on self-test (POST), as well as how, after it has been loaded, an SMP OS can detect the additional processors and assign tasks for them to execute.

This Chapter

This chapter provides a detailed description of the processor logic responsible for instruction fetch, decode and execution.

The Next Chapter

In preparation for the discussion of the processor's caches, the next chapter introduces the register set that tells the processor its rules of conduct within various areas of memory. The Memory Type and Range Registers, or MTRRs, must be programmed after startup to define the various regions of memory within the 4GB memory space and how the processor core and caches must behave when performing accesses within each region. A detailed description of the MTRRs may be found in the appendix entitled "The MTRR Registers" on page 505.

Please Note

This chapter discusses the internal architecture of the Pentium Pro processor. It must be stressed that this information is based on the information released by Intel to the general development community. Since Intel is not in the business of giving away their intellectual property, it stands to reason that they haven't revealed every detail of the processor's operation. The author has made every effort to faithfully describe the processor's operation within the bounds of the information released by Intel.

Introduction

Throughout this chapter, refer to the overall processor block diagram pictured in Figure 5-1 on page 63. Figure 24-7 on page 457 illustrates the block diagram with the MMX excution units added in.

At the heart of the processor are the execution units that execute instructions. The processor includes a fetch engine that attempts to properly predict the path of program execution and generates an on-going series of memory read operations to fetch the desired instructions.

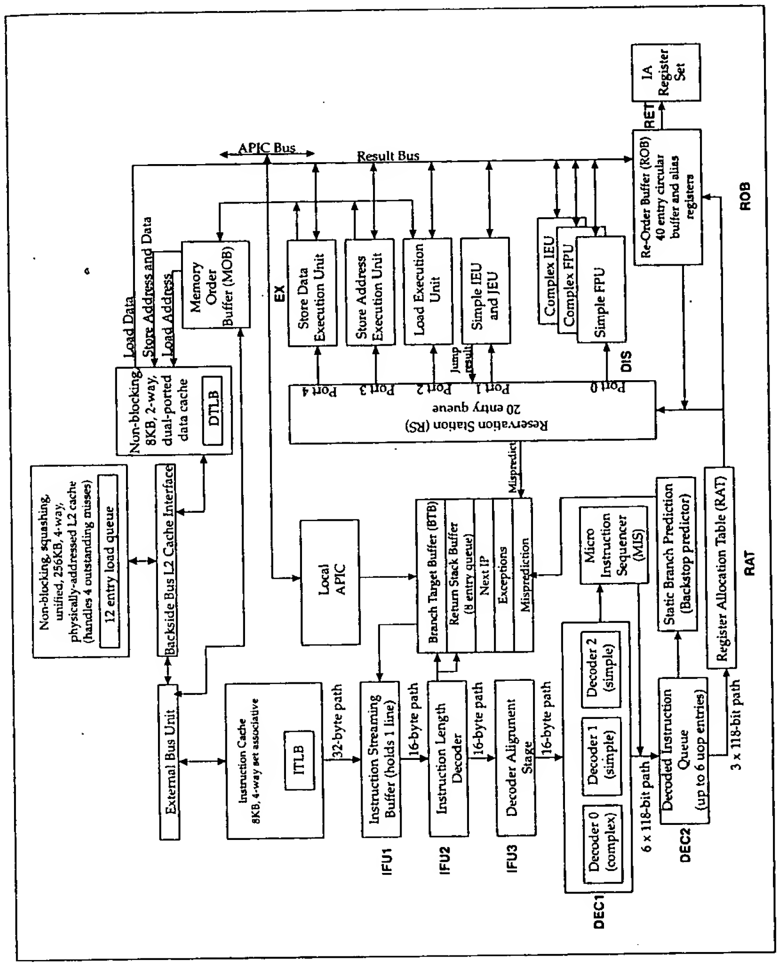
If the processor didn't include an instruction cache, each of these memory read requests would have to be issued (on the processor's external bus) to external memory. The processor would suffer severe performance degradation for two reasons:

- The clock rate at which the processor generates external bus transactions is a fraction of the internal clock rate. This results in a relatively low-speed memory read to fetch the instructions.
- Secondly, the access time of main DRAM memory is typically even slower than the bus speed. In other words, the memory injects wait states into the bus transaction until it has the requested information available.

The high-speed (e.g., 150 or 200MHz) processor execution engine would then be bound by the speed of external memory accesses. It should be obvious that it is extremely advantageous to include a very high-speed cache memory on board the processor to keep copies of recently used (and, hopefully, frequently used) information (both code and data). Memory read requests generated by the processor core are first submitted to the cache for a lookup before being propagated to the external bus (in the event of a cache miss).

The Pentium Pro processor includes both a code and a data cache (the level one, or L1, caches). In addition, it includes an L2 cache tightly coupled to the processor core via a private bus. The processor's caches are disabled at powerup time, however. In order to realize the processor's full potential, the caches must be enabled. The following section describes the enabling of the caches.

Figure 5-1: Overall Processor Block Diagram


Non-blocking, squashing, unified, 256KB, 4-way, physically-addressed L2 cache (handles 4 outstanding misses)
12 entry load queue
External Bus Unit
Backside Bus L2 Cache Interface
Non-blocking 8KB, 2-way, dual-ported data cache
DTLB
Load Data
Store Address and Data
Load Address
Memory Order Buffer (MOB)
APIC Bus
Result Bus
EX
Store Data Execution Unit
Store Address Execution Unit
Load Execution Unit
Simple IEU and JEU
Jump result
Complex IEU
Complex FPU
Simple FPU
DIS
Port 4
Port 3
Port 2
Port 1
Port 0
Reservation Station (RS) 20 entry queue
Mispredict
Instruction Cache 8KB, 4-way set associative
ITLB
Local APIC
32-byte path
IFU1
Instruction Streaming Buffer (holds 1 line)
16-byte path
IFU2
Instruction Length Decoder
16-byte path
IFU3
Decoder Alignment Stage
16-byte path
Branch Target Buffer (BTB)
Return Stack Buffer (8 entry queue)
Next IP
Exceptions
Misprediction
DEC1
Decoder 0 (complex)
Decoder 1 (simple)
Decoder 2 (simple)
Micro Instruction Sequencer (MIS)
6 x 118-bit path
DEC2
Decoded Instruction Queue (up to 6 uop entries)
Static Branch Prediction (Backstop predictor)
3 x 118-bit path
Register Allocation Table (RAT)
RAT
Re-Order Buffer (ROB) 40 entry circular buffer and alias registers
ROB
RET
IA Register Set

revealed every detail of the processor's operation. The author has made every effort to faithfully describe the processor's operation within the bounds of the information released by Intel.

Introduction

Throughout this chapter, refer to the overall processor block diagram pictured in Figure 5-1 on page 63. Figure 24-7 on page 457 illustrates the block diagram with the MMX excution units added in.

At the heart of the processor are the execution units that execute instructions. The processor includes a fetch engine that attempts to properly predict the path of program execution and generates an on-going series of memory read operations to fetch the desired instructions.

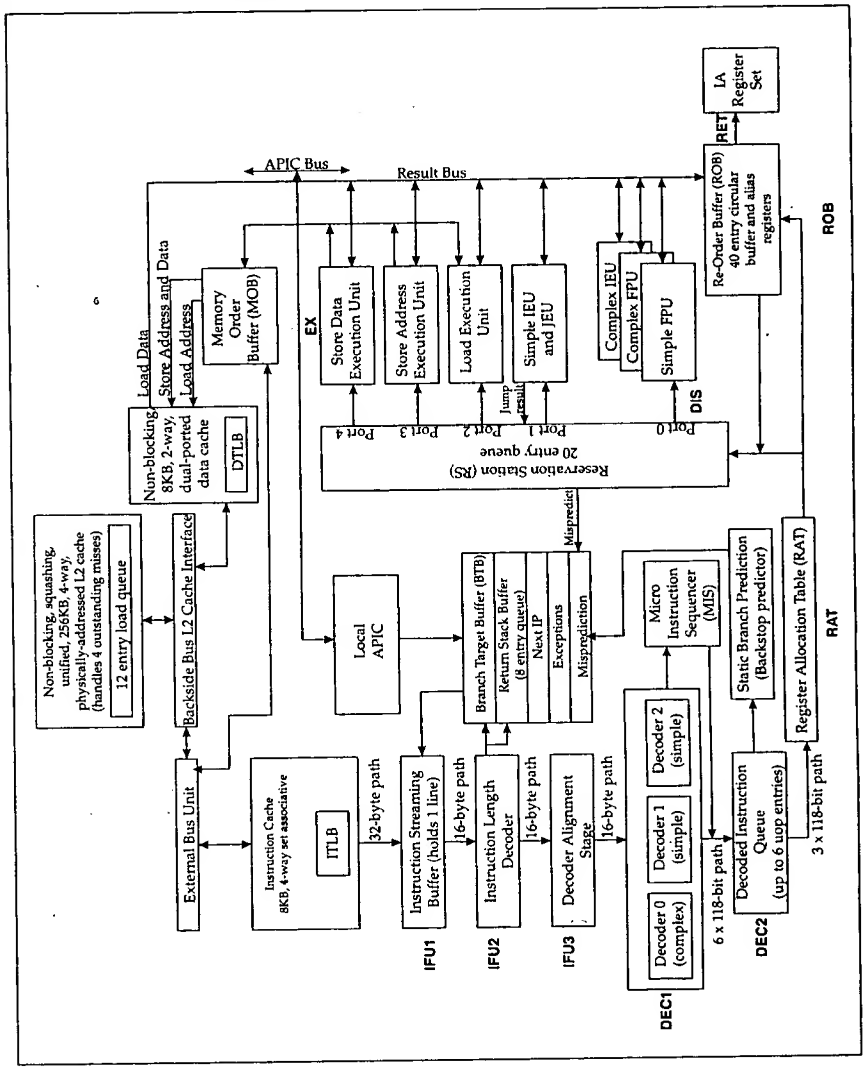
If the processor didn't include an instruction cache, each of these memory read requests would have to be issued (on the processor's external bus) to external memory. The processor would suffer severe performance degradation for two reasons:

- The clock rate at which the processor generates external bus transactions is a fraction of the internal clock rate. This results in a relatively low-speed memory read to fetch the instructions.
- Secondly, the access time of main DRAM memory is typically even slower than the bus speed. In other words, the memory injects wait states into the bus transaction until it has the requested information available.

The high-speed (e.g., 150 or 200MHz) processor execution engine would then be bound by the speed of external memory accesses. It should be obvious that it is extremely advantageous to include a very high-speed cache memory on board the processor to keep copies of recently used (and, hopefully, frequently used) information (both code and data). Memory read requests generated by the processor core are first submitted to the cache for a lookup before being propagated to the external bus (in the event of a cache miss).

The Pentium Pro processor includes both a code and a data cache (the level one, or L1, caches). In addition, it includes an L2 cache tightly coupled to the processor core via a private bus. The processor's caches are disabled at powerup time, however. In order to realize the processor's full potential, the caches must be enabled. The following section describes the enabling of the caches.

Figure 5-1: Overall Processor Block Diagram


Non-blocking, squashing, unified, 256KB, 4-way, physically-addressed L2 cache (handles 4 outstanding misses)
12 entry load queue
External Bus Unit
Backside Bus L2 Cache Interface
Non-blocking, 8KB, 2-way, dual-ported data cache
DTLB
Load Data
Store Address and Data
Load Address
Memory Order Buffer (MOB)
APIC Bus
Result Bus
EX
Instruction Cache 8KB, 4-way set associative
ITLB
32-byte path
Local APIC
Store Data Execution Unit
Store Address Execution Unit
Load Execution Unit
Simple IEU and JEU
Jump result
Complex IEU
Complex FPU
Simple FPU
Reservation Station (RS) 20 entry queue
Port 4
Port 3
Port 2
Port 1
Port 0
DIS
IFU1
Instruction Streaming Buffer (holds 1 line)
16-byte path
IFU2
Instruction Length Decoder
16-byte path
IFU3
Decoder Alignment Stage
16-byte path
Branch Target Buffer (BTB)
Return Stack Buffer (8 entry queue)
Next IP
Exceptions
Misprediction
Mispredict
DEC1
Decoder 0 (complex)
Decoder 1 (simple)
Decoder 2 (simple)
Micro Instruction Sequencer (MIS)
6 x 118-bit path
DEC2
Decoded Instruction Queue (up to 6 uop entries)
Static Branch Prediction (Backstop predictor)
3 x 118-bit path
Register Allocation Table (RAT)
RAT
Re-Order Buffer (ROB) 40 entry circular buffer and alias registers
RET
IA Register Set
ROB

Enabling the Caches

The processor's caches are disabled at startup (by reset) and are enabled when:

- CR0[CD] and CR0[NW] = 00b,
- and the targeted memory area is designated as cacheable by the MTRRs (memory type and range registers, described in the chapter entitled "Rules of Conduct" on page 119) and/or the selected page table entry (described in the chapter entitled "Paging Enhancements" on page 379). For more information on cache enabling, refer to the chapter entitled "The Processor Caches" on page 133.

Memory read requests are not issued to the code cache when:

- the caches are disabled or
- the target memory area is designated as non-cacheable by the MTRRs and/or the selected Page Table entry.

The following sections assume that the caches are enabled (via CR0) and the targeted memory area is designated cacheable (via the MTRRs and/or the selected page table entry). In this case, all code memory read requests generated by the instruction prefetcher are submitted to the code cache for a lookup. This discussion also assumes that the request results in a hit on the code cache. The code cache then supplies the requested information to the prefetcher.

Pref tch r

Issues Sequential Read Requests to Code Cache

In Figure 5-1 on page 63, the prefetcher interacts with the prefetch streaming buffer in the IFU1 pipeline stage. Although not directly pictured, it is represented by the Next IP block in the picture. The prefetcher continues issuing sequential access requests to the code cache unless one of the following events occurs:

- external interrupt.
- software exception condition.
- previously-fetched branch instruction results in a hit on the Branch Target Buffer (BTB) and the branch is predicted taken, or that static branch prediction results in a prediction of a branch to be taken.

In Intel's Pentium Pro documentation, the prefetch queue is referred to as the prefetch streaming buffer. The prefetch streaming buffer can hold 32 bytes of code (Please note that Intel's documentation states that it holds two 16-byte lines. In reality, it holds one line divided into two 16-byte blocks.). Assuming that caching is enabled, the prefetcher attempts to always keep the buffer full by issuing read requests to the code cache whenever the buffer becomes empty.

From the prefetch streaming buffer, the instructions enter the instruction pipeline. The series of stages that an instruction passes through from fetch to completion is referred to as the pipeline. The prefetch stage is the first stage of the pipeline.

Detailed Description of Prefetcher Operation

Figure 5-2 on page 66 illustrates a 64 byte area of memory (from memory location 0h through 3Fh) that contains a series of program instructions (for the time being, ignore the "S" and "C" designations). Notice that the length of the instructions vary. Some are one byte in length, some two bytes, etc. Assume that the processor's branch prediction logic predicts a branch to instruction one (the first instruction represented in grey). The prefetcher issues a request to the code cache for the line (consisting of memory locations 0-1Fh) that contains the first byte of the target instruction. These 32 bytes are loaded into the processor's prefetch streaming buffer (see Figure 5-3 on page 66).

In this example, the first 25-bytes (those that precede the branch target address) are unwanted instruction bytes and are discarded by the prefetcher. It should be noted that, although the figures illustrate the boundary between each instruction, in reality the boundaries are not marked in the prefetch streaming buffer. At this point, this is just a "raw" code stream. The boundaries will be determined and marked at a later stage of the pipeline (see "Decode Stages" on page 74).

These "raw" instructions are fed to instruction pipeline 16-bytes at a time. Until another branch is predicted, the prefetcher requests the next sequential line from the code cache each time that the streaming buffer is emptied by the processor core.

Figure 5-2: Example Instructions in Memory

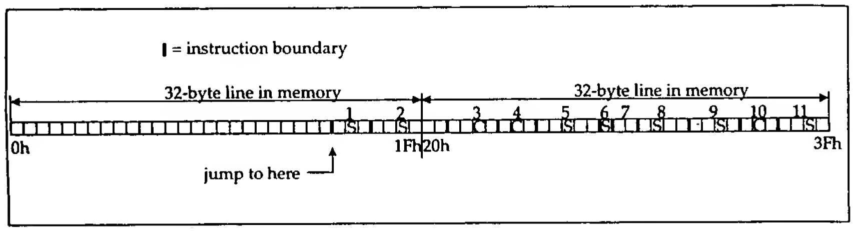
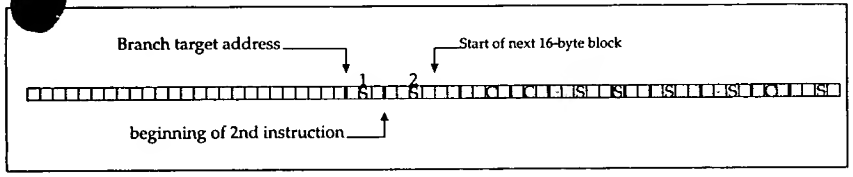


Figure 5-3: Contents of Prefetch Streaming Buffer Immediately after Line Fetched



In the example pictured in Figure 5-3 on page 66, the first 16-byte block is discarded, as are the first nine bytes of the second 16-byte block. Only instruction one is passed to the complex decoder (see Figure 5-4 on page 67) in the current clock. The complex decoder is referred to as decoder zero, while the two simple decoders are decoders one and two.

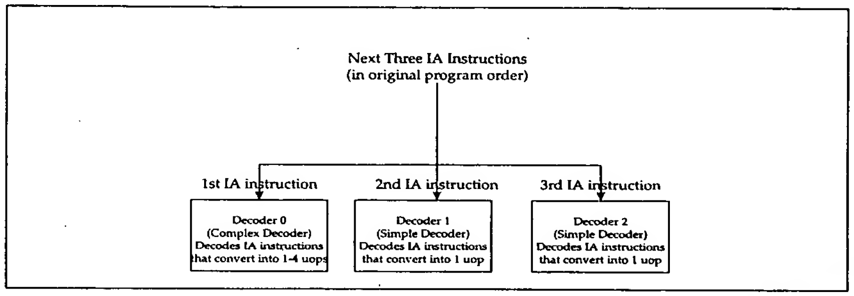
The second instruction cannot be passed to a decoder yet because it overflows into the second 16-byte block. In the next clock, the second 16-byte block is accessed, supplying the remainder of instruction two. The prefetcher always attempts to pass three instructions to decoders zero, one and two in strict program order (in the same clock). The earliest instruction in the next group of three instructions is aligned with decoder 0, while next two in-line instructions are aligned with decoders 1 and 2, respectively. The decoders are pictured in Figure 5-4 on page 67. Instruction two is therefore lined up with decoder 0, while instructions three and four are lined up with decoders 1 and 2, respectively. However, instructions 3 and 4 are both complex and cannot be handled by the simple decoders, so only instruction 2 is passed to decoder 0 in this clock.

The next three instructions that have not yet been decoded (instructions 3, 4, and 5) are aligned with decoders 0 through 2, respectively, and consist of a complex/complex/simple series. While instruction 3 can be handled by decoder 0, instruction 4 cannot be handled by decoder 1. Since instructions are always

decoded in strict program order, instruction 5 cannot be decoded before instruction 4, so only instruction 3 is passed to decoder 0 in this clock.

In the next clock, the next three instructions, (4, 5, and 6) are lined up with decoders 0 through 2, respectively. In other words, a complex/simple/simple instruction series is lined up with decoders 0 through 2, respectively. In this case, all three instructions can be decoded simultaneously during the current clock.

Figure 5-4: The Three Instruction Decoders


Next Three IA Instructions (in original program order)
1st IA instruction
2nd IA instruction
3rd IA instruction
Decoder 0 (Complex Decoder) Decodes IA instructions that convert into 1-4 uops
Decoder 1 (Simple Decoder) Decodes IA instructions that convert into 1 uop
Decoder 2 (Simple Decoder) Decodes IA instructions that convert into 1 uop

Brief Description of Pentium Pro Processor

The processor:

1. Fetches IA instructions from memory in strict program order.
2. Decodes, or translates, them (in strict program order) into one or more fixed-length RISC instructions known as micro-ops, or uops.
3. Places the micro-ops into an instruction pool in strict program order.
4. Until this point, the instructions have been kept in original program order. This part of the pipeline is known as the in-order front end. The processor then executes the micro-ops in any order possible as the data and execution units required for each micro-op become available. This is known as the out-of-order (OOO) portion of the pipeline.
5. Finally, the processor commits the results of each micro-op execution to the processor's register set in the order of the original program flow. This is the in-order read-end.

Beginning, Middle and End

In-Order Front End

In the first seven stages of the instruction pipeline (the prefetch, decode, RAT and ROB stages), the instructions are kept in strict program order. The instructions are fetched, decoded into micro-ops, stored in the instruction decode queue and are then moved into the ROB in strict program order.

Out-of-Order (OOO) Middle

Once the micro-ops are placed in the instruction pool (i.e., the ROB) in strict program order, they are executed out-of-order, one or more at a time (up to five instructions can be dispatched and start execution simultaneously), as data and execution units for each micro-op become available. As each micro-op in the instruction pool completes execution, it is marked as ready for retirement and its results are retained in the micro-op's entry in the instruction pool. Intel has implemented features that permit the processor to execute instructions out-of-order:

- **Register aliasing**. Eliminates false dependencies created by the small IA register set.
- **Non-blocking data and L2 caches**. Permits the processor to continue to make progress when cache misses occur.
- **Feed forwarding**. Result of one micro-op's execution are immediately made available to other micro-ops that require the results in order to proceed with their own execution.

In-Order Rear End

The retirement unit (in the RET1 and RET2 stages) is constantly testing the three oldest micro-ops in the instruction pool to determine when they have completed execution and can be retired in original program order. As each group of three micro-ops is retired, the results of their execution is "committed" to the processor's IA register set. The micro-ops are then deleted from the pool and the pool entries become available for assignment to other micro-ops.

Intro to th Instruction Pipeline

Figure 5-5 on page 69 illustrates the main instruction pipeline. Table 5-1 on page 69 provides a basic description of each stage, while the sections that follow the table provide a detailed description of each.

Figure 5-5: Instruction Pipeline

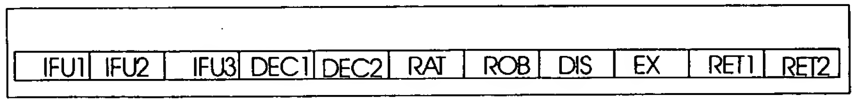


Table 5-1: Pipeline Stages

Stage	Description
IFU1	**Instruction Fetch Unit stage 1.** Load a 32-byte line from the code cache into the prefetch streaming buffer.
IFU2	**Instruction Fetch Unit stage 2.** Mark instruction boundaries. In the IFU2 stage, two operations are performed simultaneously: • The boundaries between instructions within a 16-byte block are identified. Intel documents frequently refer to this as a line of information, but the term line is typically applied to the size of a block of information in a cache. To avoid possible confusion, the author therefore has decided not to refer to this as a line. • If any of the instructions within the 16-byte block are branches, the memory addresses that they were fetched from are presented to the BTB (Branch Target Buffer) for branch prediction. Refer to "Description of Branch Prediction" on page 108.
IFU3	**Instruction Fetch Unit stage 3.** Align the instructions for presentation to the appropriate decoders (see Figure 5-4 on page 67) in the next stage.
DEC1	**Decode stage 1.** Translate (i.e., decode) the instructions into the corresponding micro-op(s). Up to three instructions can be decoded simultaneously (if they are aligned with a decoder that can handle the instruction; see Figure 5-4 on page 67).
DEC2	**Decode Stage 2.** Pass the micro-ops to the decoded instruction queue. Since some IA instructions (e.g., string operations) translate into rather large streams of micro-ops and the decoded instruction queue can only accept six micro-ops per clock, it should be noted that this stage may have to be repeated a number of times (i.e, it may last for a number of processor clock ticks).

Table 5-1: Pipeline Stages (Continued)

Stage	Description
RAT	**Register Alias Table and Allocator stage.** The processor determines if a micro-op references any source operands. If it does, the processor determines if the source operand should be taken from a real IA register or from an entry in the ROB. The latter case will be true if a micro-op previously placed in the ROB (from earlier in the code stream) will, when executed, produce the result required as a source operand by this micro-op. In this case, the source field in this micro-op is adjusted to point to the respective ROB entry. As an example, a micro-op earlier in the code stream may, when executed, place a value in the EAX register. In reality, when the processor executes this micro-op, it places the value, not in the EAX register, but rather in the ROB entry occupied by the micro-op. A micro-op later in the code stream that requires the contents of EAX as a source would be adjusted in the RAT stage to point to the result field of the ROB entry occupied by the earlier micro-op, rather than to the real EAX register.
ROB	**ReOrder Buffer (ROB) stage.** Move micro-ops from the RAT/Allocator stage to the next sequential three ROB entries at the rate of three per clock. If all of the data required for execution of a micro-op is available and an entry is available in the RS (Reservation Station) queue, also pass a copy of the micro-op to the RS. The micro-op then awaits the availability of the appropriate execution unit.
DIS	**Dispatch stage.** If not already done in the previous stage (because all of the data required by the micro-op was not available or an entry wasn't available in the RS), copy the micro-op from the ROB to the RS, and then dispatch it to the appropriate execution unit for execution.
EX	**Execution stage.** Execute micro-op. The number of clocks that an instruction takes to complete execution is instruction dependent. Most micro-ops can be executed in one clock.
RET1	**Retirement stage 1.** When a micro-op in the ROB has completed execution, all conditional branches earlier in the code stream have been resolved, and it is determined that the micro-op should have been executed, it is marked as ready for retirement.

Table 5-1: Pipeline Stages (Continued)

Stage	Description
RET2	**Retirement stage 2. Retire instruction.** When the previous IA instruction has been retired and all of the micro-ops associated with the next IA instruction have completed execution, the real IA register(s) affected by the IA instruction's execution are updated with the instruction's execution result (this is also referred to as "committing" the results to the machine state) and the micro-op is deleted (i.e., retired) from ROB. Up to three micro-ops are retired per clock in strict program order.

In-Order Front End

Instruction Fetch Stages

The first three stages of the instruction pipeline are the instruction fetch stages. The Intel documentation states that this takes 2.5 clock periods, but it doesn't define which clock edges within that 2.5 clock period define the boundaries between the three stages.

IFU1 Stage: 32-Byte Line Fetched from Code Cache

The instruction pipeline feeds from the bottom end of the prefetch streaming buffer, requesting 16 bytes at a time. When the buffer is emptied, the prefetch logic issues a request (during the IFU1 stage) to the code cache for the next sequential line. The prefetcher issues a 32-byte-aligned address to the code cache and a 32-byte block of "raw" code (the next sequential line) is loaded into the prefetch streaming buffer by the code cache.

IFU2 Stage: Marking Boundaries and Dynamic Branch Prediction

During the IFU2 stage, the boundaries between instructions within the first 16-byte block are identified and marked. In addition, if any of the instructions are branches, the memory addresses that they were fetched from (for up to four branches within the 16-byte block) are presented to the branch target buffer (BTB) for branch prediction. For a detailed discussion of dynamic branch prediction, refer to the section entitled "Description of Branch Prediction" on page 108. Assuming that none of the instructións are branches or that none of the branches are predicted taken, code fetching will continue along the same sequential memory path.

IFU3 Stage: Align Instructions for Delivery to Decoders

Refer to Figure 5-6 on page 73. The processor implements three decoders that are used to translate the variable-length IA instructions into fixed-length RISC instructions referred to as micro-ops. Each micro-op is 118 bits in length and is referred to as a triadic micro-op because it includes three elements (in addition to the RISC instruction itself):

- two sources
- one destination

Intel does not document the format of the micro-ops.

The first of the three decoders (decoder 0) is classified as a complex decoder and can translate any IA instruction that translates into between one and four micro-ops. The second and third decoders (decoders 1 and 2) are classified as simple decoders and can only translate IA instructions that translate into single micro-ops (note that most of the IA instructions translate into one micro-op). In general:

- Simple IA instructions of the register-register form convert to a single micro-op.
- Load instructions (i.e., memory data reads) convert to a single micro-op.
- Store instructions (i.e., memory data writes) convert to two micro-ops.
- Simple read/modify instructions convert into two micro-ops.
- Simple instructions of the register-memory form convert into two or three micro-ops.
- MMX instructions are simple.
- Simple read/modify/write instructions convert into four micro-ops.

Appendix D of Intel Application Note AP-526 (order Number 242816) contains the number of micro-ops that each IA instruction converts to.

In the IFU3 stage, the processor aligns the first of the next three sequential IA instructions (within the 16-byte block) with the complex decoder and, if it translates into no more than four micro-ops, delivers it to the decoder for translation into micro-ops. In addition, if the second and possibly the third of the three instructions are simple (i.e., translate into a single micro-op each), they can simultaneously be delivered to the simple decoders (decoders 1 and 2) for translation into micro-ops.

The complexity of each IA instruction and the order of the instructions dictates how many instructions can be delivered to the decoders (somewhere between one and three) for translation in a single clock. Consider the example scenarios described in Table 5-2 on page 73.

Figure 5-6: The Three Instruction Decoders

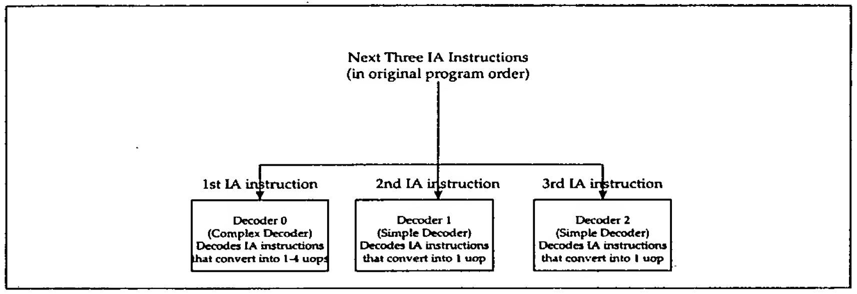

Next Three IA Instructions (in original program order)
1st IA instruction
2nd IA instruction
3rd IA instruction
Decoder 0 (Complex Decoder) Decodes IA instructions that convert into 1-4 uops
Decoder 1 (Simple Decoder) Decodes IA instructions that convert into 1 uop
Decoder 2 (Simple Decoder) Decodes IA instructions that convert into 1 uop

Table 5-2: Examples of IA Instruction Delivery to the Three Decoders

If next 3 Sequential IA Instructions in 16-byte Block are:	Instructions delivered to decoders in single clock
Simple, simple, simple	three
Simple, simple, complex	first two
Simple, complex, simple	first one
Simple, complex, complex	first one
Complex, simple, simple	three
Complex, simple, complex	first two
Complex, complex, simple	first one
Complex, complex, complex	first one

Decode Stages

After the IA instructions are aligned with the three decoders, they are submitted to one or more of the decoders (see "IFU3 Stage: Align Instructions for Delivery to Decoders" on page 72). There are two decode pipeline stages, DEC1 and DEC2, taking a total of 2.5 clocks to complete. The following sections discuss these two stages.

DEC1 Stage: Translate IA Instructions into Micro-Ops

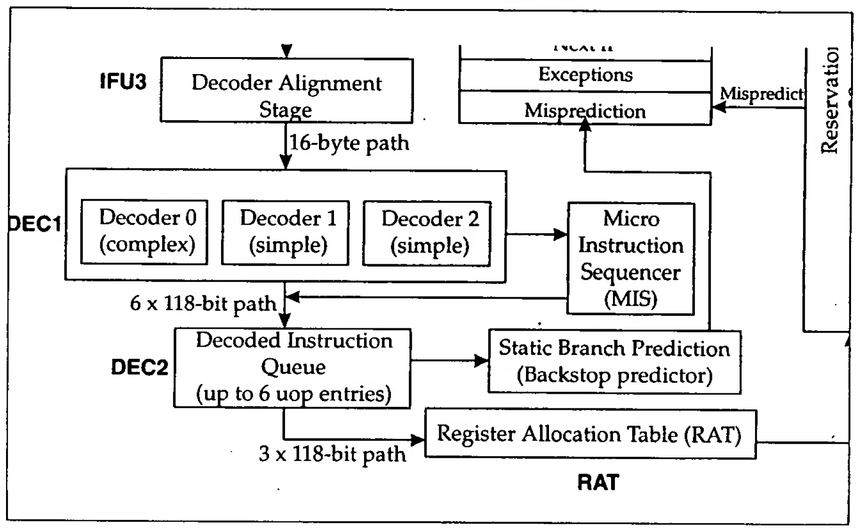
In the DEC1 stage, between one and three IA instructions are submitted to decoders 0 through 2 for translation into micro-ops. The complex decoder can decode any IA instruction that is not greater than seven bytes in length and that translates into no more than four micro-ops. The simple decoders can decode any IA instruction that is not greater than seven bytes in length and that translates into a single micro-op. Most IA instructions are categorized as simple.

The best case throughput scenario is when a complex instruction is presented to decoder 0, and two simple instructions are simultaneously submitted to decoders 1 and 2. In this case, up to six micro-ops can be created in one clock (up to four from the complex instruction and one each for the two simple instructions). In the DEC2 stage (see "DEC2 Stage: Move Micro-Ops to ID Queue" on page 75), the micro-ops created are placed into the ID (instruction decode) queue in the same order as the IA instructions that they were translated from (i.e., in program order).

Micro Instruction Sequencer (MIS)

Refer to Figure 5-7 on page 75. Some IA instructions translate into more than four micro-ops and therefore cannot be handled by decoder 0. These instructions are submitted to the micro instruction sequencer (MIS) for translation. Essentially, the MIS is a microcode ROM that contains the series of micro-ops (five or more) associated with each very complex IA instruction. Some instructions (e.g., string operations) may translate into extremely large micro-op sequences. The micro-ops produced are placed into the ID queue in the DEC2 stage.

Figure 5-7: Decoders and the Micro Instruction Sequencer (MIS)



DEC2 Stage: Move Micro-Ops to ID Queue

Two operations occur in the decode 2 stage:

1. The micro-ops produced by the decoders (or by the MIS) are placed in the ID queue in original program order. The best-case scenario is one where, in the DEC1 stage, the complex decoder is presented with an IA instruction that translates into four micro-ops and the two simple decoders are simultaneously presented with simple IA instructions that each create a single micro-op. In this case, six micro-ops are the loaded into the ID queue simultaneously (note that the path between the DEC1 and DEC2 stages is 6 x 118-bits wide).
2. If any of the micro-ops in the ID queue represent branches, static branch prediction (for more information, refer to "Static Branch Prediction" on page 113) is applied to predict if the branch will be taken when executed or not.

Queue Micro-Ops for Placement in Pool. In the DEC2 stage, the micro-ops created by the three decoders or by the MIS are stored in the ID queue in original program order. The ID queue can hold up to six micro-ops and is pictured in Figure 5-7 on page 75.

Second Chance for Branch Prediction. As the micro-ops are deposited in the ID queue, the processor checks to see if any are branches. If any are, the processor's static branch prediction algorithm is applied to predict whether, when executed, the branch will be taken. For a detailed discussion of static branch prediction, refer to the section entitled "Static Branch Prediction" on page 113.

RAT Stage: Overcoming the Small IA Register Set

The IA register set only includes 16 registers that can be used to store data. They are the eight floating-point registers (FP0 through FP7) and the EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP registers. The small number of IA registers restricts the programmer to keeping a very small number of data variables close at hand in the registers. When these registers are all in use and the programmer needs to retrieve another data variable from memory, the programmer is forced to first copy the current contents of one of the IA registers back to memory before loading that register with new data from another area of memory. As an example, the following code fragment saves off the data in EAX before loading a new value into EAX:

```
mov  [mem1], EAX
mov  EAX,[mem2]
```

Due to the small register set, this sequence is very common in x86 programs. Ordinarily, the processor could not execute these two instructions simultaneously (because EAX must first be saved before it can be loaded with the new value). This is commonly referred to as a false register dependency that prevents the simultaneous execution of the two instructions.

However, the processor is designed to use another, hidden register as the register to be written by the second instruction and can therefore execute both instructions simultaneously. In other words, at the same time that the first instruction is reading the value from EAX to write into memory, the second instruction can read from memory and write into a hidden register rather than the real EAX register. The value in the hidden EAX can be copied into the actual EAX at a later time.

As another example, consider the following code fragment:

```
mov  eax,17
add  mem,eax
mov  eax,3
add  eax,ebx
```

Ordinarily, these four instructions would have to be executed in serial fashion, one after the other. The Pentium Pro can execute the first and the third instructions simultaneously, moving the values into two, separate hidden registers rather than into the eax register. Likewise, the second and the fourth instructions can then be executed simultaneously. The eax reference in the second instruction is changed to point to the hidden register that was loaded with the value 17 by the first instruction. The eax reference in the fourth instruction is changed to point to the hidden register that was loaded with the value 3 by the third instruction.

The processor includes a set of 40 hidden registers that can be used in lieu of any of the eight general purpose registers or any of the eight floating-point registers. This eliminates many false dependencies between instructions, thereby permitting simultaneous execution and resulting in better performance. During the register alias table (RAT) stage, the processor selects which of the 40 surrogate registers will actually be used by the micro-op when it is executed. The surrogate registers reside in the ROB (discussed in the next section).

ReOrder Buffer (ROB) Stage

After the micro-ops are produced by the decoders and have had their source fields adjusted in the RAT stage, they are placed in the 40-deep ROB in strict program order, at the rate of up to three per clock. Once placed in this instruction pool, the RS (reservation station) can copy multiple micro-ops from the pool (up to five simultaneously) and queue them up for delivery to the appropriate execution units. When micro-ops in the ROB have completed execution and are retired (described later), they are deleted from the ROB. These ROB entries are then available to move new micro-ops to the ROB from the ID queue. The sections that follow provide a detailed description of the ROB.

Instruction Pool (ROB) is a Circular Buffer

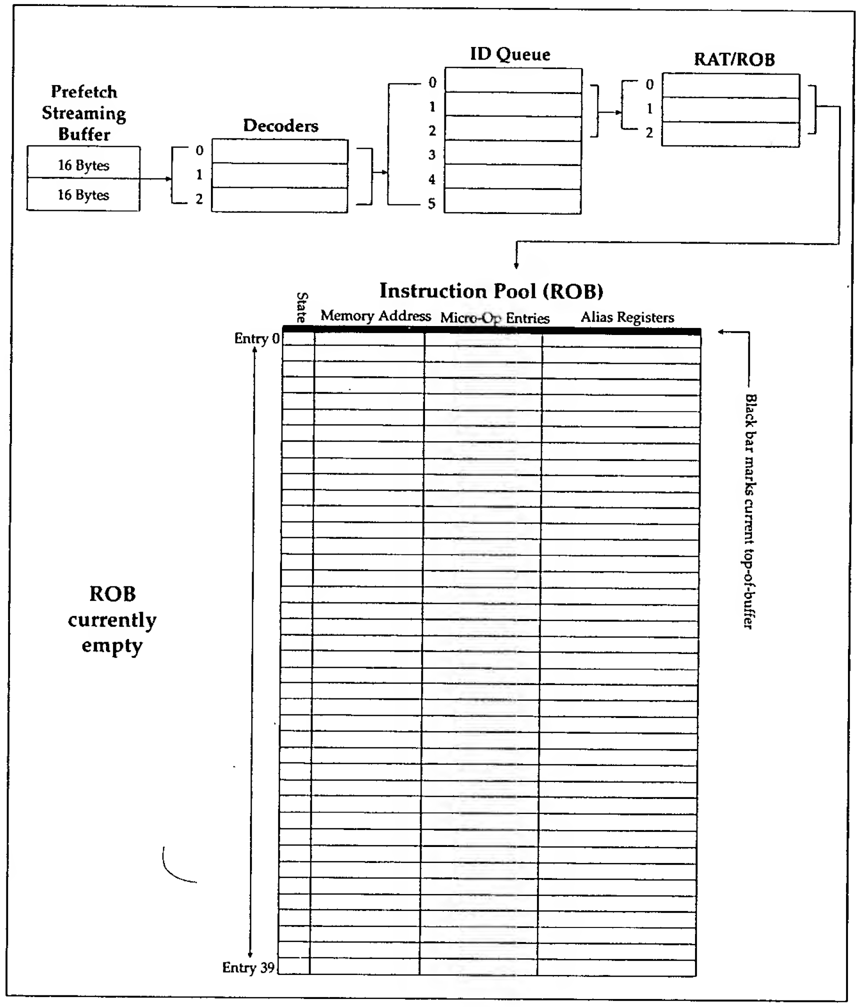
Refer to Figure 5-8 on page 79. During each clock, between one and three micro-ops (depending on how many micro-ops are currently in the ID queue) are moved from the top three locations (zero through two) of the ID queue to the Register Alias Table (RAT) stage. As stated earlier (see "RAT Stage: Overcoming the Small IA Register Set" on page 76), the processor adjusts the micro-ops' source fields to point to the correct registers (i.e., one of the real IA registers, or one of the 40 hidden registers in the ROB).

The ROB is implemented as circular buffer with 40 entries. A micro-op and, after it has completed execution, the result of its execution, are stored in each ROB entry. Initially, the ROB is empty and the start-of-buffer pointer and the end-of-buffer pointer both point to entry 0. As IA instructions are decoded into micro-ops, the micro-ops (up to three per clock) are placed in the ROB starting at entry 0 in strict program order. As additional micro-ops are produced by the decoders, the end-of-buffer pointer is incremented once for each micro-op. At a given instant in time, the end-of-buffer pointer points to where the next micro-op decoded will be stored, while the start-of-buffer pointer points to the oldest micro-op in the buffer (corresponding to the earliest instruction in the program).

Later, in the retirement stage, the retirement logic will retire (remove) the three oldest micro-ops from the pool and will increment the start-of-buffer pointer by three. The three pool entries that were occupied by the three micro-ops just retired are then available for new micro-ops.

To summarize, the logic is always adding new micro-ops to the end-of-buffer, incrementing the end-of-buffer pointer, and is also removing the oldest micro-ops from the start-of-buffer and incrementing the start-of-buffer pointer.

Figure 5-8: The ROB, the RS and the Execution Units



Out-of-Order (OOO) Middle

After the micro-ops are placed in the instruction pool (i.e., the ROB), the RS can copy multiple micro-ops from the pool in any order and dispatch them to the appropriate execution units for execution. The criteria for selecting a micro-op for execution is that the appropriate execution unit and all necessary data items required by the micro-op are available—the micro-ops do not have to be executed in any particular order. Once executed, the results of the micro-op's execution are stored in the ROB entry occupied by the micro-op.

In-Order Rear End (RET1 and RET2 Stages)

As already described in the previous sections, the processor fetches and decodes IA instructions in program order. The micro-ops produced by the decoders are placed into the ID queue in program order and are moved into the ROB in original program order. Once placed in the ROB, however, the micro-ops are dispatched and executed in any order possible. The execution results are stored in the respective ROB entries (rather than in the processor's actual register set).

When all upstream branches have been resolved (i.e., executed) and it has been established that a downstream micro-op that has already completed execution should have been executed, the micro-op is marked at ready for retirement (in the RET1 stage).

The retirement logic constantly checks the status of the oldest three micro-ops in the ROB (at the start-of-buffer) to determine when all three of them have been marked as ready for retirement. The micro-ops are then retired (in the RET2 stage), three at a time, in original program order. As each is retired, the micro-op's execution result is copied into the processor's real register set from the ROB entry and the respective ROB entry is then deleted.

Three Scenarios

To explain the relationship of—

- the prefetch streaming buffer,
- the decoders,
- the ID queue,
- the RAT,
- the ROB,
- the reservation station (RS),
- and the execution units,

the following sections have been included. They describe three example scenarios:

- The first assumes that reset has just been removed from the processor.
- The second assumes that the processor's caches have just been enabled.
- The third assumes that the processor's caches have been enabled for some time and the code cache contains lines of code previously fetched from memory.

Scenario One: Reset Just Removed

Immediately after reset is removed, the following conditions exist:

- Code cache and L2 cache are empty and caching is disabled.
- Prefetch streaming buffer is empty.
- ID queue is empty.
- ROB is empty.
- Reservation station (RS) is empty.
- All execution units are idle.
- The CS:IP registers are pointing to memory location FFFFFFF0h.

Starvation!

The ROB and the RS are empty, so the processor's execution units have no instructions to execute.

First Instruction Fetch

Because caching is disabled (CR0[CD] and CR0[NW] are set to 11b), the prefetcher bypasses the L1 code cache and the L2 cache and submits its memory read request directly to the external bus interface (this occurs in the IFU1 stage). This read request has the following characteristics:

- Memory address is FFFFFFF0h, identifying the quadword (group of eight locations) consisting of memory locations FFFFFFF0h through FFFFFFF7h.
- When caching is disabled or when fetching code from an area of memory designated as non-cacheable (by the MTRRs or the selected page table entry), the prefetcher always requests eight bytes of information. In other words, it takes advantage of the full width of the processor's external data bus (64-bits).

First Memory Read Bus Transaction

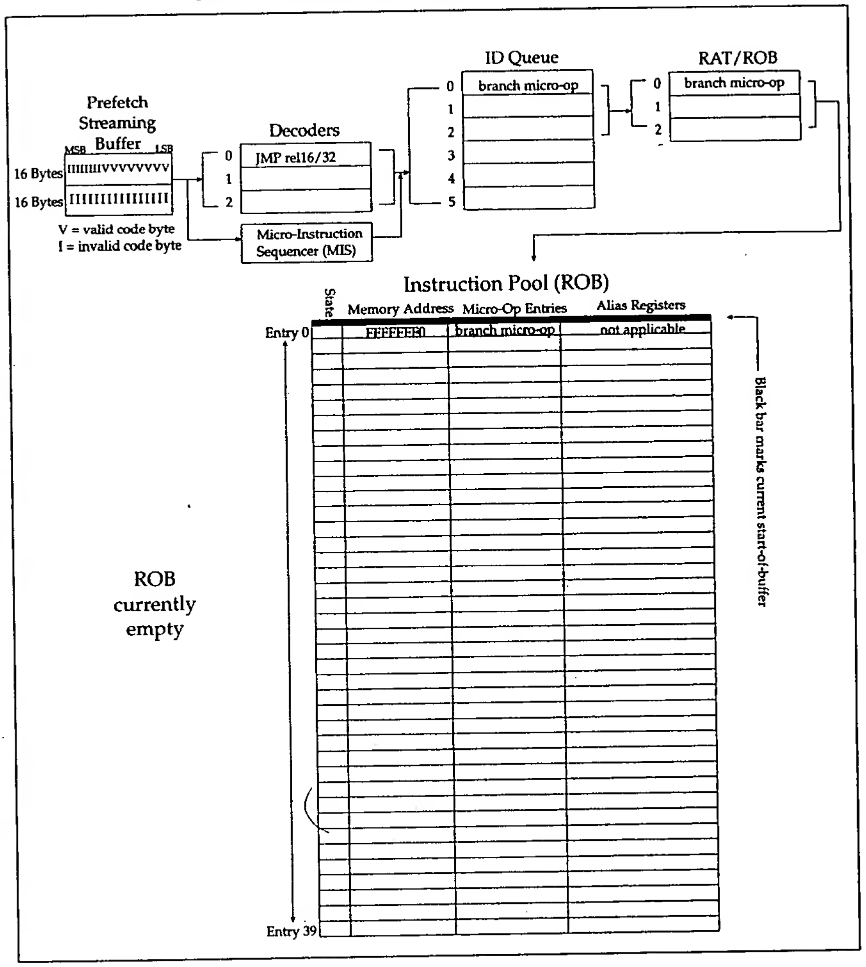
In response to this 8-byte memory read request, the external bus interface arbitrates for ownership of the external bus and then initiates an 8-byte memory read. *Please note that, even though caching is disabled, the current implementations of the Pentium Pro processor initiate a 32-byte rather than an 8-byte read. The critical quadword (the one that starts at 0FFFFFFF0h) is returned by the boot ROM, followed by the remaining three quadwords that comprise the line. The processor passes the first quadword into the prefetch streaming buffer and discards the remaining three quadwords. When the prefetcher issues the request for the next sequential 8-bytes, the processor initiates another 32-byte read for the same line, keeps the first quadword and discards the next three returned. Whether or not future versions of the Pentium Pro will also indulge in this inefficient behavior is unknown at this time.*

During this period of time, the processor core starvation continues (because the instructions have not percolated down through the instruction pipeline stages to be decoded and executed).

Eight Bytes Placed in Prefetch Streaming Buffer

Refer to Figure 5-9 on page 83. The addressed memory device presents the eight bytes back to the processor. The processor latches the data and places it in the first eight locations of the prefetch streaming buffer. The prefetcher immediately issues a request for the next sequential group of eight memory locations, FFFFFFF8h through FFFFFFFFh and the bus interface initiates another external bus transaction.

Figure 5-9: Scenario One Example—Reset Just Removed

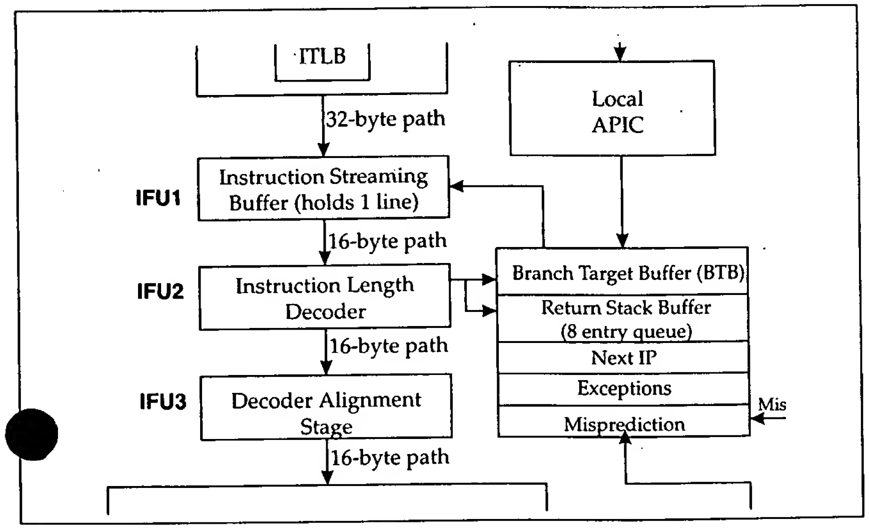


Instruction Boundaries Marked and BTB Checked

Refer to Figure 5-10 on page 84. In the IFU2 stage, the instruction boundaries within the eight-byte block are marked. In addition, if any of the instructions are branches (up to four of them), the memory address that it was fetched from is submitted to the BTB for a lookup. The BTB, a cache, was cleared by reset, so the lookup results in a BTB miss. In other words, the BTB has no history on the branch and therefore cannot predict whether or not it will be taken when it arrives at the jump execution unit (JEU) and is executed. Sequential fetching continues.

Realistically speaking, the first instruction fetched from the power-on restart address (FFFFFFF0h) is almost always an unconditional branch to an address lower in memory (because FFFFFFF0h is only 16 locations from the top of memory space).

Figure 5-10: The IFU2 Stage


ITLB
32-byte path
Local APIC
IFU1
Instruction Streaming Buffer (holds 1 line)
16-byte path
IFU2
Instruction Length Decoder
Branch Target Buffer (BTB)
Return Stack Buffer (8 entry queue)
Next IP
Exceptions
Misprediction
Mis
16-byte path
IFU3
Decoder Alignment Stage
16-byte path

Between One and Three Instructions Decoded into Micro-Ops

Refer to Figure 5-11 on page 86 and to Figure 5-9 on page 83. In the DEC1 stage, the first three IA instructions (assuming that there are three embedded within the first eight bytes) are submitted to the three decoders for translation into

micro-ops. The best case throughput scenario would be a simple/simple/simple or a complex/simple/simple sequence. This would produce between three and six micro-ops in one clock cycle.

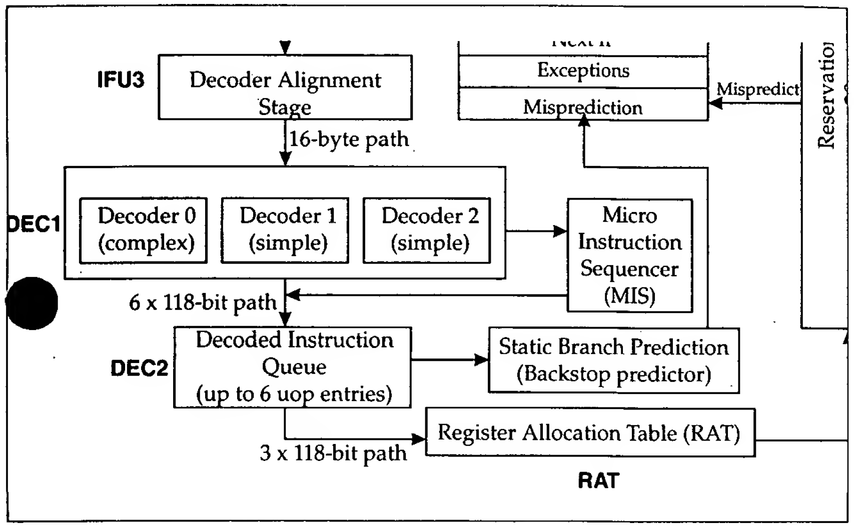
In the DEC2 stage, the micro-ops are forwarded to the ID queue. In addition, if any of the micro-ops are branches, the static branch prediction logic decides whether or not to alter the fetch stream (i.e., to predict whether or not the branch will be taken when it arrives at the jump execution unit (JEU) and is executed).

As mentioned earlier, the first instruction fetched after power-up is almost always an unconditional branch to an address lower in memory (the entry point of the system's power-on self-test, or POST). If this is the case, the static branch prediction logic predicts whether the branch will be taken when it is executed by the JEU (for more information, refer to "Static Branch Prediction" on page 113).

Assuming that the first instruction is an unconditional branch, it is predicted as taken (in the DEC2 stage). This has the following effects:

- The instructions in the IFU1 through DEC2 stages (i.e., in the prefetch streaming buffer and the ID queue) are flushed (the unconditional branch remains in the ID queue, however). The flushing of the instructions in the first five stages causes a "bubble" in the pipeline, resulting in a temporary decrease in performance.
- The branch target address is supplied to the prefetcher and is used to fetch the next instruction.
- Both the predicted branch target address and the fall-through address accompany the branch until it is executed by the JEU.

Figure 5-11: Decoders and the Micro Instruction Sequencer (MIS)



Source Operand Location Selected (RAT)

Refer to Figure 5-11 on page 86 and to Figure 5-9 on page 83. The branch instruction advances to the RAT stage where the processor determines if the micro-op references any source operands. If it does, the processor determines if the source operand should be supplied from a real IA register or from an entry in the ROB. The latter case will be true if a micro-op previously placed in the ROB (from earlier in the code stream) will, when executed, produce the result required as a source operand for this micro-op. In this case, the source field in this micro-op is adjusted to point to the respective ROB entry.

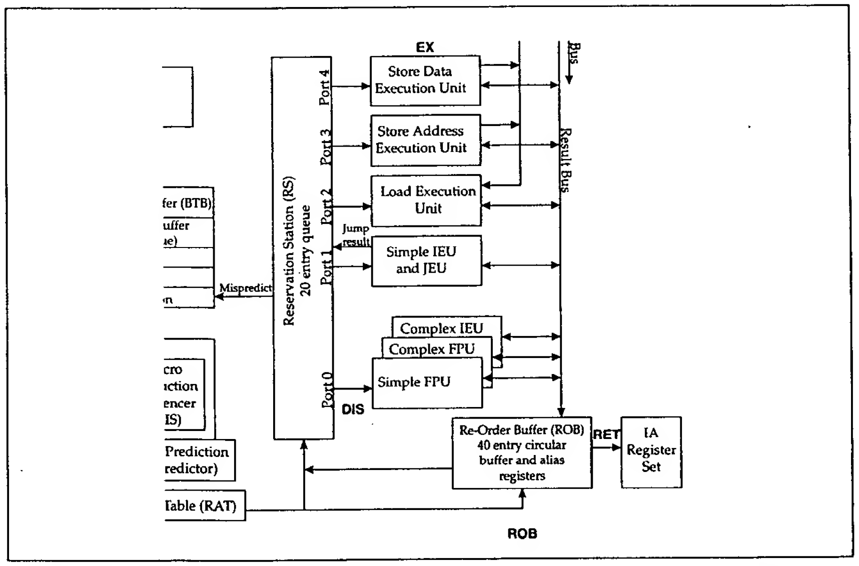
As an example, a micro-op earlier in the code stream may, when executed, place a value in the EAX register. In reality, when the processor executes this micro-op, it places the value, not in the EAX register, but rather in the ROB entry occupied by the micro-op. A micro-op later in the code stream that requires the contents of EAX as a source would be adjusted in the RAT stage to point to the result field of the ROB entry occupied by the earlier micro-op.

Micro-Ops Advanced to ROB and RS

Refer to Figure 5-12 on page 87. The reservation station (RS) is the gateway to the processor's execution units (see "The ROB, the RS and the Execution Units" on page 79). Basically, it is a queuing mechanism that, according to the Intel documentation, can queue up to 20 micro-ops to be forwarded to execution units as they become available. The documents don't define whether it is implemented as one 20-entry queue common to all five execution ports or as five separate queues (one for each execution port) each with four entries.

In the ROB stage, up to three micro-ops per clock are moved from the ID queue to the ROB in strict program order and are placed in the next three available ROB entries. In this example scenario, the branch micro-op in the ID queue is simultaneously moved to ROB entry zero and to the RS (because the RS has queue space available). In the event that there are no open RS queue entries, the micro-op is moved from the RAT to the ROB, but is not copied into the RS from the ROB until room becomes available (as the respective execution unit completes the execution of another micro-op).

Figure 5-12: The ReOrder Buffer (ROB) and The Reservation Station (RS)



Micro-Ops Dispatched for Execution

Refer to Figure 5-12 on page 87. The JEU is currently idle, so the branch micro-op is immediately dispatched from the RS to the JEU for execution during the EX (execution) stage and begins execution. In ROB entry zero, the branch micro-op is marked as executing (see Figure 5-9 on page 83).

Micro-Ops Executed

In this example, the micro-op is an unconditional branch, so the branch is taken when executed. The RS provides this feedback to the BTB where an entry is made indicating that the branch should be predicted taken the next time that is fetched. In addition, the branch target address is also stored in the BTB entry so that it will know where to branch to.

Since the branch was correctly predicted (by the static branch prediction logic during the DEC2 stage) and prefetching altered accordingly, the correct stream of instructions are behind the branch in the pipeline (albeit with a bubble in between). If the branch had been incorrectly predicted, the instructions that were prefetched after the branch, translated and placed in the ROB would be incorrect and must be flushed, along with those in pipeline stages earlier than the ROB (i.e., the RAT, ID queue, and prefetch streaming buffer).

Result to ROB Entry (and other micro-ops if necessary)

The branch micro-op has completed execution and its result is stored in the same ROB entry as the micro-op. The micro-op is marked completed in its respective ROB entry. Another micro-op currently awaiting execution in the ROB or RS may require the result produced by this micro-op. In this case, the result produced by the just-completed micro-op is now available to the stalled micro-op. That micro-op can then be scheduled for execution.

Micro-op Ready for Retirement?

It must be noted that micro-ops in the ROB are executed out-of-order (i.e., not in the original IA instruction order) as the required execution unit and any required data operands become available. A micro-op later in the program can therefore complete execution before instructions earlier in the program flow.

Consider the situation where there is a branch micro-op in the ROB from earlier in the program flow that has not yet been executed. When that branch is executed it may or may not alter the program flow. If it branches to a different area of memory, the micro-ops currently in the ROB that were prefetched from the

other path should not have been fetched, decoded and executed. Any micro-op already executed from the mispredicted path must therefore be deleted along with any results that it produced (in other words, as if it had never been executed).

A micro-op is not ready for retirement until it is certain that no branches from earlier in the program are currently in the ROB that, when executed, might result in a branch that would preclude the execution of this micro-op. When this condition is met, the branch micro-op in ROB entry zero is marked (in the RET1 stage) as ready for retirement.

Micro-Op Retired

In order to have the program execute in the order originally intended by the programmer, micro-ops must be retired in original program order. In other words, a micro-op that has completed execution and been marked as ready for retirement cannot be retired until all micro-ops that precede it in the program have been retired.

Retiring a micro-op means to permit its execution result to affect the processor's state (i.e., the contents of its real register set). Consider the following example:

```
add  edx,ebx
mov  eax,[0100]
cmp  eax,edx
jne  loop
```

The instructions that comprise this code fragment must alter and/or observe the processor's real register set in the following order:

1. edx must be changed to reflect the result of the addition.
2. eax must then be loaded with the four bytes from memory locations 00000100h through 00000103h in the data segment.
3. The values in edx and eax are then compared, altering the appropriate condition bits in the processor's EFLAGS register.
4. The conditional jump instruction then checks the EFLAGS[EQUAL] bit to determine whether or not to branch.

The process of retiring a micro-op involves:

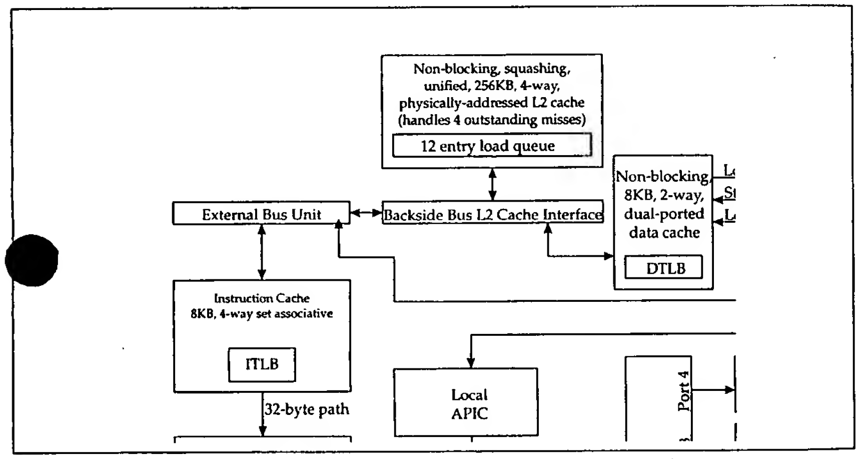
- Permitting its result (currently stored only in the ROB entry) to be copied to the processor's real register set.
- Deleting the micro-op from the ROB
- The ROB entry then becomes available to accept another micro-op.

The processor is capable of retiring up to three micro-ops per clock during the RET2 stage.

Scenario Two: Processor's Caches Just Enabled

Assuming that the processor's caches have just been enabled (by clearing both CR0[CD] and CR0[NW] to zero), the caches are currently empty. Also assume that the processor's MTRR registers have been set up to define some memory as cacheable. When the prefetcher requests the next quadword, the quadword address is submitted to the code cache for a lookup. This results in a miss and the code cache issues a read request to the L2 cache (see Figure 5-13 on page 90) for the 32-byte code line that contains the requested quadword. This results in an L2 cache miss and the cache line request is issued to the processor's bus interface unit. The bus interface arbitrates for ownership of the external bus and initiates a 32-byte memory read request from main memory. The 32-byte line is returned to the processor in toggle-mode order (see "Toggle Mode Transfer Order" on page 171). The line is placed in the L2 cache and is also passed to the code cache where another copy is made. In addition, the line is immediately passed from the code cache to the prefetch streaming buffer. From this point, the processing of the code proceeds as already discussed starting in the section entitled "Instruction Boundaries Marked and BTB Checked" on page 84.

Figure 5-13: Code Cache and L2 Cache Miss

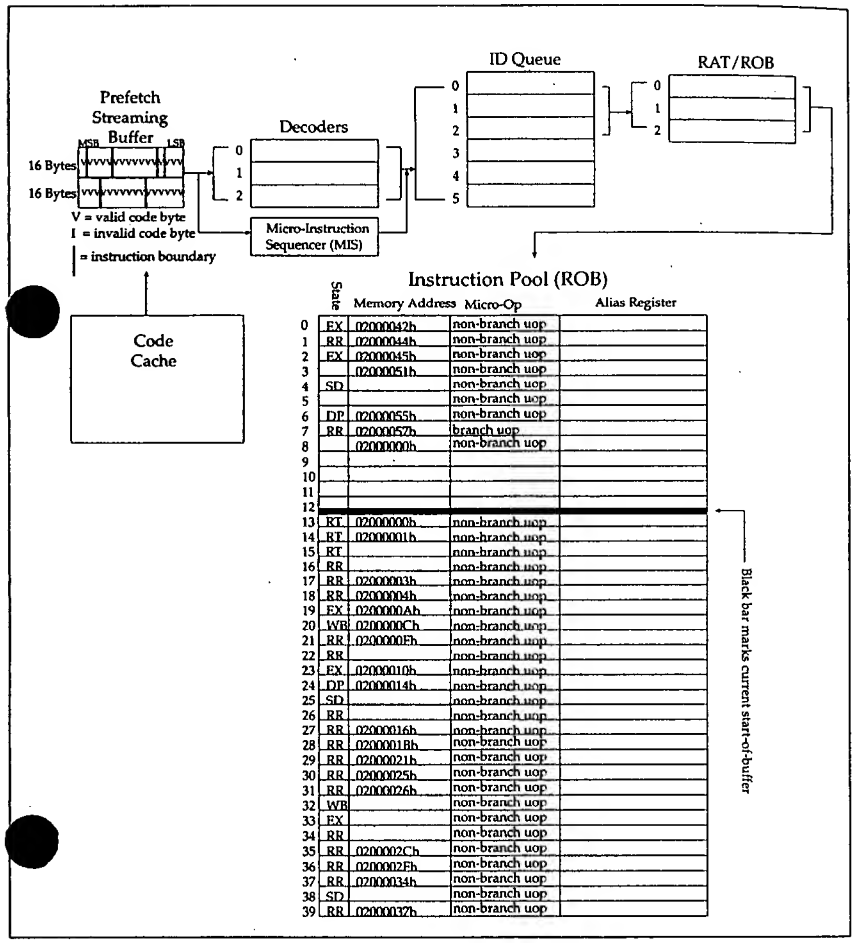

Non-blocking, squashing, unified, 256KB, 4-way, physically-addressed L2 cache (handles 4 outstanding misses)
12 entry load queue
External Bus Unit
Backside Bus L2 Cache Interface
Non-blocking, 8KB, 2-way, dual-ported data cache
DTLB
Instruction Cache 8KB, 4-way set associative
ITLB
32-byte path
Local APIC
Port 4

Sc nario Three: Caches Enabled for Some Time

In this scenario, the processor's caches have been enabled for some time and the processor's instruction pipeline, including the ID queue, the ROB and the RS currently contain a stream of micro-ops in various stages of completion. Refer to Figure 5-14 on page 92 during this discussion.

Micro-ops (such as those illustrated in Figure 5-14 on page 92) are always placed in the ROB in original program order. The instruction in ROB entry 13 is the oldest instruction currently in the ROB, while the instruction in entry eight in the youngest. Instructions must be retired in strict program order, from oldest to youngest. The basic format of an entry (note that Intel does not provide this level of information, so this table is based on hopefully intelligent speculation) is introduced in Table 5-3 on page 93, while an entry-by-entry description is provided in Table 5-4 on page 94. Note that the micro-ops in Table 5-4 on page 94 are listed in original program order.

Figure 5-14: Scenario Three



	State	Memory Address	Micro-Op	Alias Register
0	EX	02000042h	non-branch uop	
1	RR	02000044h	non-branch uop	
2	EX	02000045h	non-branch uop	
3		02000051h	non-branch uop	
4	SD		non-branch uop	
5			non-branch uop	
6	DP	02000055h	non-branch uop	
7	RR	02000057h	branch uop	
8		02000000h	non-branch uop	
9				
10				
11				
12				
13	RT	02000000h	non-branch uop	
14	RT	02000001h	non-branch uop	
15	RT		non-branch uop	
16	RR		non-branch uop	
17	RR	02000003h	non-branch uop	
18	RR	02000004h	non-branch uop	
19	EX	0200000Ah	non-branch uop	
20	WB	0200000Ch	non-branch uop	
21	RR	0200000Fh	non-branch uop	
22	RR		non-branch uop	
23	EX	02000010h	non-branch uop	
24	DP	02000014h	non-branch uop	
25	SD		non-branch uop	
26	RR		non-branch uop	
27	RR	02000016h	non-branch uop	
28	RR	0200001Bh	non-branch uop	
29	RR	02000021h	non-branch uop	
30	RR	02000025h	non-branch uop	
31	RR	02000026h	non-branch uop	
32	WB		non-branch uop	
33	EX		non-branch uop	
34	RR		non-branch uop	
35	RR	0200002Ch	non-branch uop	
36	RR	0200002Fh	non-branch uop	
37	RR	02000034h	non-branch uop	
38	SD		non-branch uop	
39	RR	02000037h	non-branch uop	

Table 5-3: Basic Description of a ROB Entry

Element	Description
State	A micro-op in the ROB can be in one of the following states: • **SD**: scheduled for execution. Indicates that the micro-op has been queued in the RS, but has not yet been dispatched to an execution unit. • **DP**: indicates that the micro-op is at the head of the RS dispatch queue for its respective execution port and is being dispatched to the respective execution unit for execution. • **EX**: the micro-op is currently being executed by its respective execution unit. • **WB**: the micro-op has completed execution and its results are being written back to the micro-op's ROB entry. In addition, if any other micro-ops are stalled waiting for the result, the result is forwarded to the stalled micro-op(s). • **RR**: the micro-op is ready for retirement. • **RT**: the micro-op is being retired.
Memory address	Indicates the start memory address of the IA instruction that generated the micro-op(s).
Micro-op	The micro-op is either a branch or a non-branch instruction.
Alias register	If a micro-op references one of the IA registers, the reference is re-directed to one of the 40 registers contained within the ROB (each ROB entry contains a register field that can store a value up to 80-bits wide (a floating-point value would be 80-bits wide, while integer values may be one, two, or four bytes wide).

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries

Entry	Explanation
13	This is the oldest micro-op in the ROB and, along with the next two micro-ops, is being retired in the current clock. The new start-of-buffer will then be 16 and entries 13, 14 and 15 will be available for new micro-ops (see Figure 5-15 on page 102). Entry 13 was the only micro-op decoded from the 1-byte IA instruction that was fetched from memory address 02000000h. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000001h (see entry 14).
14	This micro-op (and those in entries 15 and 16) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. Along with entries 13 and 15, it is being retired in the current clock. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
15	This micro-op (and those in entries 14 and 16) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. Along with entries 13 and 14, it is being retired in the current clock. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
16	This micro-op (and those in entries 14 and 15) was decoded from the 2-byte IA instruction that was fetched from memory locations 02000001h and 02000002h. It is ready for retirement and will be retired in the next clock (see Figure 5-16 on page 103) along with entries 17 and 18. The new start-of-buffer will then be 19 and entries 16, 17 and 18 will be available for new micro-ops. Entry 16 is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000003h (see entry 17).
17	This micro-op was the only one decoded from the 1-byte IA instruction that was fetched from memory location 02000003h. It is ready for retirement (see Figure 5-16 on page 103) and will be retired in the next clock along with entries 16 and 18. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000004h (see entry 18).
18	This micro-op was the only one decoded from the 6-byte IA instruction that was fetched from memory locations 02000004h through 02000009h. It is ready for retirement (see Figure 5-16 on page 103) and will be retired in the next clock along with entries 16 and 17. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Ah (see entry 19).

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
19	This micro-op was the only one decoded from the 2-byte IA instruction that was fetched from memory locations 0200000Ah and 0200000Bh. It is currently being executed and, depending on the type of micro-op, may take one or more clocks to complete execution. When it completes execution, its state will change from EX to RR. It will be retired when: • it has completed execution • its results have been written back to entry 19 • it and the micro-ops in entries 20 and 21 are ready for retirement • the micro-ops in entries 13 through 18 have been retired It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Ch (see entry 20).
20	This micro-op was the only one decoded from the 3-byte IA instruction that was fetched from memory locations 0200000Ch through 0200000Eh. It has completed execution and its results are currently being written back to entry 20 (the micro-op is currently in the writeback stage). It will then be ready for retirement, but will not be retired until entries 19 and 21 are also ready for retirement (see entry 19). At that point, entries 19 through 21 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200000Fh (see entry 21).
21	This micro-op (and the one in entry 22) was decoded from the one-byte IA instruction that was fetched from memory location 0200000Fh. It is ready for retirement, but will not be retired until entries 19 and 20 are also ready for retirement (see entry 19). At that point, entries 19 through 21 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000010h (see entry 23).
22	This micro-op (and the one in entry 21) was decoded from the one-byte IA instruction that was fetched from memory location 0200000Fh. It will be retired when: • it has completed execution • its results have been written back to entry 22 • it and the micro-ops in entries 23 and 24 are ready for retirement • the micro-ops in entries 13 through 21 have been retired It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000010h (see entry 23).
23	This micro-op was the only one decoded from the 4-byte IA instruction that was fetched from memory locations 02000010h through 02000013h. It is still executing (and it may take more than one clock). After execution, it will enter the writeback stage where its execution results will be written back to entry 23. It will then be ready for retirement, but will not be retired until entries 22 and 24 are also ready for retirement (see entry 22). At that point, entries 22 through 24 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000014h (see entry 24).

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
24	This micro-op and the ones in entries 25 and 26 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is currently being dispatched to its respective execution unit and will begin execution in the next clock. After execution, it will enter the writeback stage where its execution results will be written back to entry 24. It will then be ready for retirement, but will not be retired until entries 22 and 23 are also ready for retirement (see entry 22). At that point, entries 22 through 24 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).
25	This micro-op and the ones in entries 24 and 26 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is currently in the RS and is scheduled for dispatch to its respective execution unit. After execution, it will enter the writeback stage where its execution results will be written back to entry 25. It will be retired when: • it has completed execution • its results have been written back to entry 25 • it and the micro-ops in entries 26 and 27 are ready for retirement • the micro-ops in entries 13 through 24 have been retired It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).
26	This micro-op and the ones in entries 24 and 25 were decoded from the 2-byte IA instruction that was fetched from memory locations 02000014h and 02000015h. It is ready for retirement, but will not be retired until entries 25 and 27 are also ready for retirement (see entry 25). At that point, entries 25 through 27 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000016h (see entry 27).
27	This micro-op was the only one decoded from the 5-byte IA instruction that was fetched from memory locations 02000016h through 0200001Ah. It is ready for retirement, but will not be retired until entries 25 and 26 are also ready for retirement (see entry 25). At that point, entries 25 through 27 will be retired. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 0200001Bh (see entry 28).
28	This micro-op was the only one decoded from the 6-byte IA instruction that was fetched from memory locations 0200001Bh through 02000020h. It is ready for retirement, as are those in entries 29 and 30. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000021h (see entry 29).
29	This micro-op was the only one decoded from the 4-byte IA instruction that was fetched from memory locations 02000021h through 02000024h. It is ready for retirement, as are those in entries 28 and 30. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000025h (see entry 30).

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
30	This micro-op was the only one decoded from the 1-byte IA instruction that was fetched from memory location 02000025h. It is ready for retirement, as are those in entries 28 and 29. It will be retired after the micro-ops in entries 13 through 27 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000026h (see entry 31).
31	This micro-op (and those in entries 32 through 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is ready for retirement and will be retired when: • the micro-op in entry 32 has written back its results to entry 32 and is ready for retirement • the micro-op in entry 33 has completed execution, has written its results back to entry 33 and is ready for retirement • the micro-ops in entries 13 through 30 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).
32	This micro-op (and those in entries 31, 33 and 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is writing its results back to entry 32 during the current clock and will be retired when: • it has written back its results to entry 32 and is ready for retirement • the micro-op in entry 33 has completed execution, has written its results back to entry 33 and is ready for retirement • the micro-ops in entries 13 through 30 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).
33	This micro-op (and those in entries 31, 32 and 34) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is executing and will be retired when: • It has completed execution and has written back its results to entry 33 and is ready for retirement • the micro-op in entry 32 has written its results back to entry 32 and is ready for retirement • the micro-ops in entries 13 through 30 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).
34	This micro-op (and those in entries 31, 32 and 33) was decoded from the 6-byte IA instruction that was fetched from memory locations 02000026h through 0200002Bh. It is ready for retirement, as are the micro-ops in entries 35 and 36. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Ch (see entry 35).

Table 5-4: *Description of Figure 5-14 on page 92 ROB Entries (Continued)*

Entry	Explanation
35	This micro-op was decoded from the 3-byte IA instruction that was fetched from memory locations 0200002Ch through 0200002Eh. It is ready for retirement, as are the micro-ops in entries 34 and 36. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 0200002Fh (see entry 36).
36	This micro-op was decoded from the 5-byte IA instruction that was fetched from memory locations 0200002Fh through 02000033h. It is ready for retirement, as are the micro-ops in entries 34 and 35. It will be retired when the micro-ops in entries 13 through 33 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000034h (see entry 37).
37	This micro-op (and the one in entry 38) was decoded from the 3-byte IA instruction that was fetched from memory locations 02000034h through 02000036h. It is ready for retirement and will be retired (along with entries 38 and 39) when: • the micro-op in entry 38 has been dispatched, executed, its results have been written back to entry 37 and it's ready for retirement. • the micro-ops in entries 13 through 36 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000037h (see entry 39).
38	This micro-op (and the one in entry 37) was decoded from the 3-byte IA instruction that was fetched from memory locations 02000034h through 02000036h. It is currently scheduled for dispatch to its respective execution unit. It will be retired (along with entries 37 and 39) when: • it has been dispatched, executed, its results have been written back to entry 37 and it's ready for retirement. • the micro-ops in entries 13 through 36 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000037h (see entry 39).
39	This micro-op was decoded from the 11-byte IA instruction that was fetched from memory locations 02000037h through 02000041h. It is ready for retirement and will be retired (along with entries 37 and 38) when: • entry 38 has been dispatched, executed, its results have been written back to entry 38and it's ready for retirement. • the micro-ops in entries 13 through 36 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000042h (see entry 0).

Table 5-4: *Description of Figure 5-14 on page 92 ROB Entries (Continued)*

Entry	Explanation
0	This micro-op was decoded from the 2-byte IA instruction that was fetched from memory locations 02000042h through 02000043h. It is currently executing and will be retired (along with entries 1 and 2) when: • it has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000044h (see entry 1).
1	This micro-op was decoded from the 1-byte IA instruction that was fetched from memory location 02000044h. It is ready for retirement and will be retired (along with entries 0 and 2) when: • the micro-op in entry 0 has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000045h (see entry 2).
2	This micro-op was decoded from the 12-byte IA instruction that was fetched from memory locations 02000045h through 02000050h. It is currently executing and will be retired (along with entries 0 and 1) when: • the micro-op in entry 0 has completed execution, its results have been written back to entry 0 and it's ready for retirement. • the micro-op in entry 2 has completed execution, its results have been written back to entry 2 and it's ready for retirement • the micro-ops in entries 13 through 39 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000051h (see entry 3).

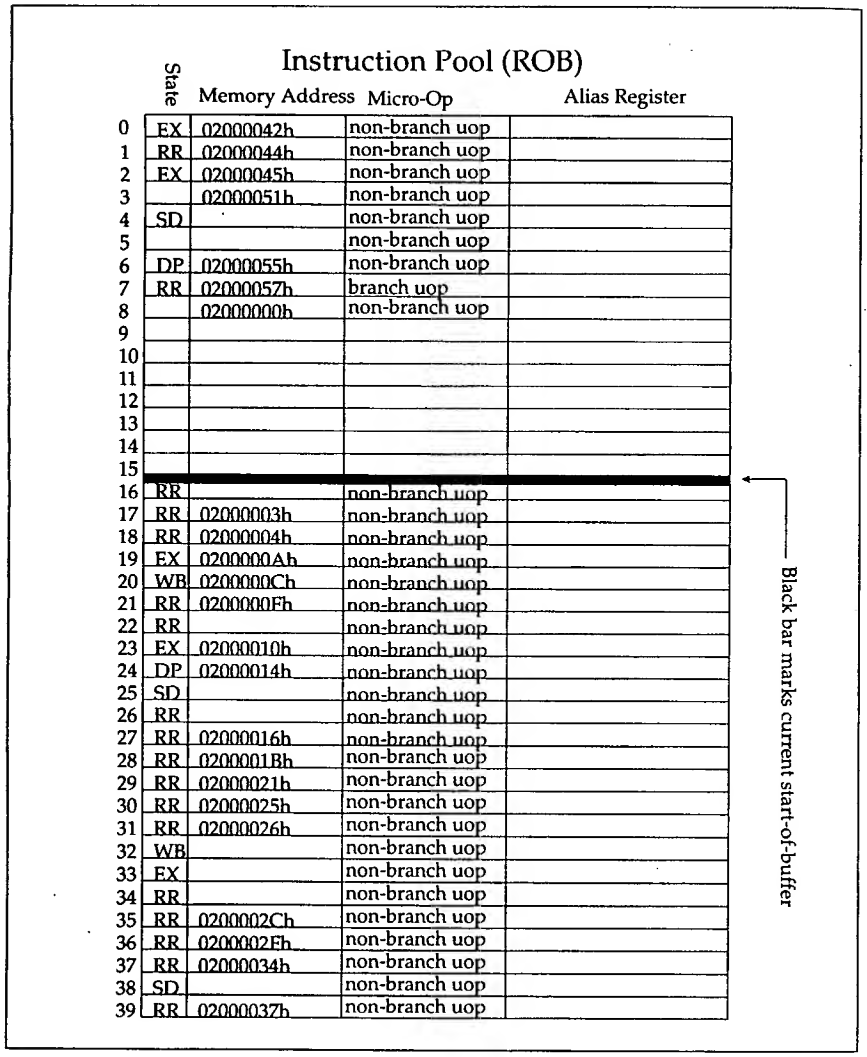
Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
3	This micro-op (and the ones in entries 4 and 5) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is has not yet entered the RS and will be retired (along with entries 4 and 5) when: • the micro-op in entry 3 has been scheduled for dispatch (i.e., it has entered the RS), dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. • the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement • the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. • the micro-ops in entries 13 through 2 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).
4	This micro-op (and the ones in entries 3 and 5) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is currently in the RS and has been scheduled for dispatch to its respective execution unit. It will be retired (along with entries 3 and 5) when: • the micro-op in entry 3 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. • the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement • the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. • the micro-ops in entries 13 through 2 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).
5	This micro-op (and the ones in entries 3 and 4) was decoded from the 4-byte IA instruction that was fetched from memory locations 02000051h through 02000054h. It is has not yet entered the RS and will be retired (along with entries 3 and 4) when: • the micro-op in entry 3 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 3 and it's ready for retirement. • the micro-op in entry 4 has dispatched, executed, its results have been written back to entry 4 and it's ready for retirement • the micro-op in entry 5 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 5 and it's ready for retirement. • the micro-ops in entries 13 through 2 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000055h (see entry 6).

Table 5-4: Description of Figure 5-14 on page 92 ROB Entries (Continued)

Entry	Explanation
6	This micro-op was decoded from the 2-byte IA instruction that was fetched from memory locations 02000055h and 02000056h. It is being dispatched to its respective execution unit. It will be retired (along with entries 7 and 8) when: • the micro-op in entry 6 has been executed, its results have been written back to entry 6 and it's ready for retirement. • the micro-op in entry 8 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 8 and it's ready for retirement. • the micro-ops in entries 13 through 5 have been retired. It is not a branch micro-op and therefore did not alter program flow. The next IA instruction was fetched from memory address 02000057h (see entry 7).
7	This branch micro-op was decoded from the IA instruction that was fetched from memory starting at location 02000057h. It will be retired (along with entries 6 and 8) when: • the micro-op in entry 6 has been executed, its results have been written back to entry 6 and it's ready for retirement. • the micro-op in entry 8 has been scheduled for dispatch, dispatched, executed, its results have been written back to entry 8 and it's ready for retirement. • the micro-ops in entries 13 through 5 have been retired. It is a branch micro-op, was predicted taken and therefore altered program flow. The next IA instruction was fetched from memory address 02000000h (see entry 8).
8	This was the only micro-op decoded from the 1-byte IA instruction that was fetched from memory address 02000000h. It is not a branch micro-op, so it did not alter program flow. The next IA instruction was fetched from memory address 02000001h.
9	empty
10	empty
11	empty
12	empty

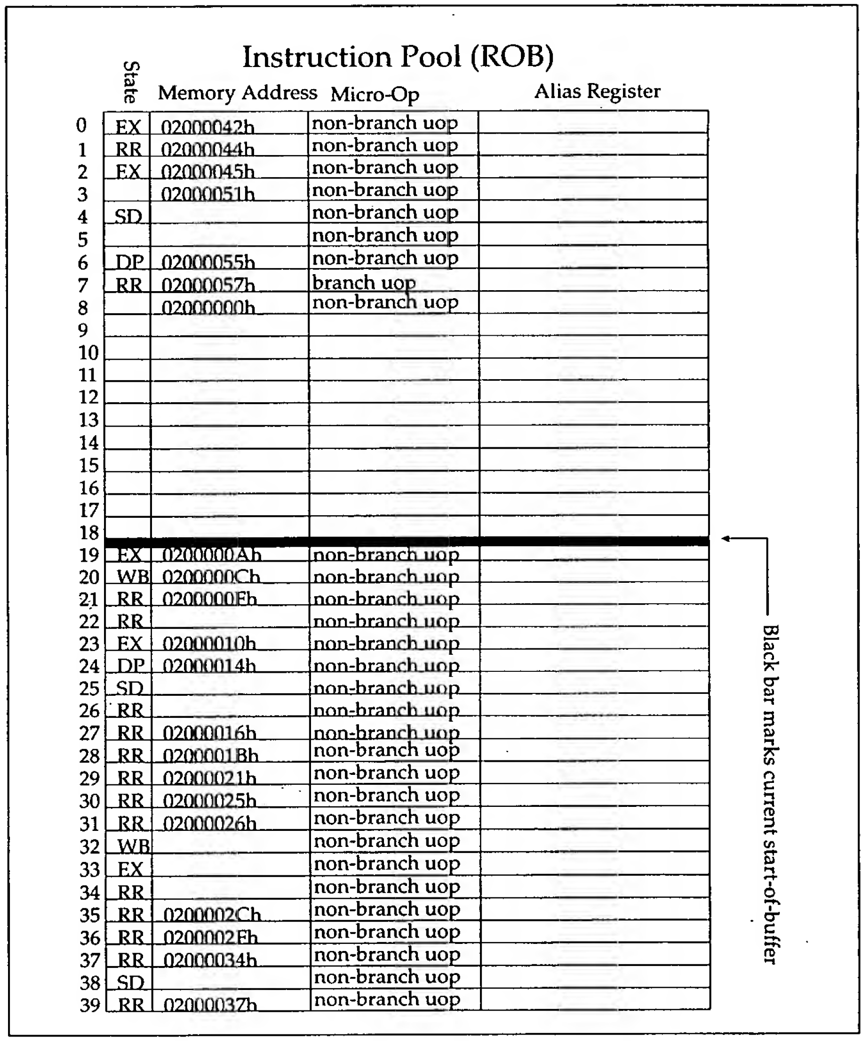
Figure 5-15: Micro-Ops in Entries 13, 14 and 15 Have Been Retired



Instruction Pool (ROB)

	State	Memory Address	Micro-Op	Alias Register
0	EX	02000042h	non-branch uop	
1	RR	02000044h	non-branch uop	
2	EX	02000045h	non-branch uop	
3		02000051h	non-branch uop	
4	SD		non-branch uop	
5			non-branch uop	
6	DP	02000055h	non-branch uop	
7	RR	02000057h	branch uop	
8		02000000h	non-branch uop	
9				
10				
11				
12				
13				
14				
15				
16	RR		non-branch uop	
17	RR	02000003h	non-branch uop	
18	RR	02000004h	non-branch uop	
19	EX	0200000Ah	non-branch uop	
20	WB	0200000Ch	non-branch uop	
21	RR	0200000Fh	non-branch uop	
22	RR		non-branch uop	
23	EX	02000010h	non-branch uop	
24	DP	02000014h	non-branch uop	
25	SD		non-branch uop	
26	RR		non-branch uop	
27	RR	02000016h	non-branch uop	
28	RR	0200001Bh	non-branch uop	
29	RR	02000021h	non-branch uop	
30	RR	02000025h	non-branch uop	
31	RR	02000026h	non-branch uop	
32	WB		non-branch uop	
33	EX		non-branch uop	
34	RR		non-branch uop	
35	RR	0200002Ch	non-branch uop	
36	RR	0200002Eh	non-branch uop	
37	RR	02000034h	non-branch uop	
38	SD		non-branch uop	
39	RR	02000037h	non-branch uop	

Figure 5-16: Micro-Ops in Entries 16, 17 and 18 Have Been Retired



Instruction Pool (ROB)

	State	Memory Address	Micro-Op	Alias Register
0	EX	02000042h	non-branch uop	
1	RR	02000044h	non-branch uop	
2	EX	02000045h	non-branch uop	
3		02000051h	non-branch uop	
4	SD		non-branch uop	
5			non-branch uop	
6	DP	02000055h	non-branch uop	
7	RR	02000057h	branch uop	
8		02000000h	non-branch uop	
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19	EX	0200000Ah	non-branch uop	
20	WB	0200000Ch	non-branch uop	
21	RR	0200000Fh	non-branch uop	
22	RR		non-branch uop	
23	EX	02000010h	non-branch uop	
24	DP	02000014h	non-branch uop	
25	SD		non-branch uop	
26	RR		non-branch uop	
27	RR	02000016h	non-branch uop	
28	RR	0200001Bh	non-branch uop	
29	RR	02000021h	non-branch uop	
30	RR	02000025h	non-branch uop	
31	RR	02000026h	non-branch uop	
32	WB		non-branch uop	
33	EX		non-branch uop	
34	RR		non-branch uop	
35	RR	0200002Ch	non-branch uop	
36	RR	0200002Eh	non-branch uop	
37	RR	02000034h	non-branch uop	
38	SD		non-branch uop	
39	RR	02000037h	non-branch uop	

Memory Data Accesses—Loads and Stores

During the course of a program's execution, a number of memory data reads (i.e., loads) and memory data writes (i.e., stores) may be performed.

Handling Loads

Refer to Figure 5-17 on page 105. An IA load instruction decodes into a single load micro-op that specifies the address to be read from and the number of bytes to read from memory starting at that address. The load (i.e., a memory data read) micro-op is executed by the load execution unit (connected to port 2 on the RS). Unless prevented from doing so (by defining an area of memory such that speculative execution is not permitted; for more information see "Rules of Conduct" on page 119), loads can be executed speculatively in any order. In other words, a load micro-op from later in the program flow can be executed before one that occurs earlier in the program flow. A load instruction flows through the normal instruction pipeline until it is dispatched to the load execution unit. It has then entered the load pipeline stages. They are illustrated in Figure 5-18 on page 105 and detailed in Table 5-5 on page 106.

When executed, the load address is always compared to those of stores currently-posted in the posted-write buffer. If any of the stores in the posted-write buffer occur earlier in the program flow then the load and the store has updated the data to be read by the load, then the store buffer supplies the data for the read. This is referred to as store, or feed forwarding. For more information about loads, refer to "L1 Data Cache" on page 143.

Figure 5-17: Load and Store Execution Units

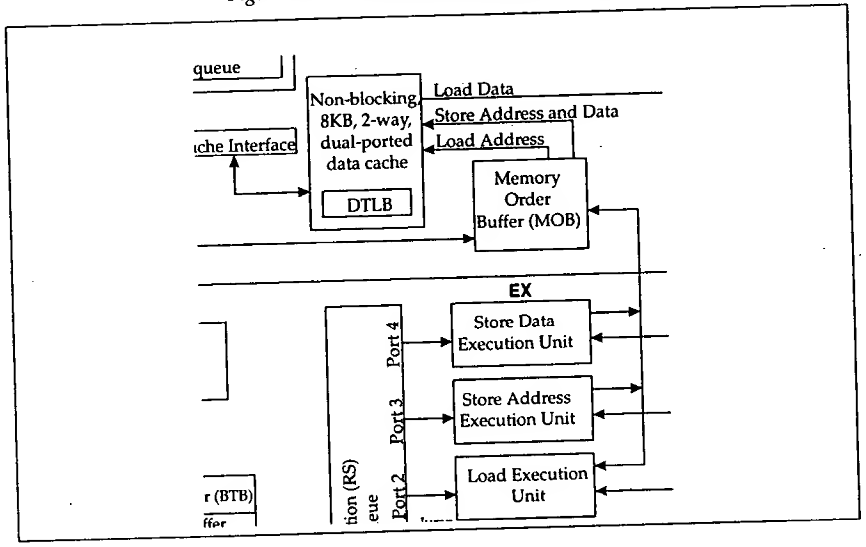


Figure 5-18: Load Pipeline Stages

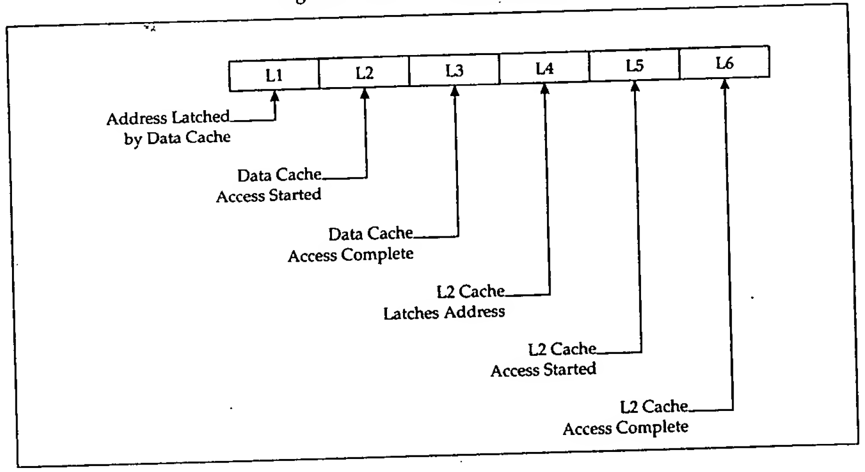


Table 5-5: Load Pipeline Stages

Stage(s)	Description
L1	During the L1 stage, the memory address for the load is translated from the linear to the physical address by the data TLB at the front end of the data cache.
L2-L3	During the L2 and L3 stages, the L1 data cache is accessed. • If this results in a hit on the L1 data cache, the data is supplied to the RS and the micro-op's ROB entry in the L3 stage. The micro-op then proceeds to the RR stage (see RR entry in this table). • If there is a miss on the L1 data cache, the instruction enters the L4 stage (see next entry).
L4-L6	In the event that the load misses the L1 data cache, the load proceeds to the L4, L5 and L6 stages where the L2 cache lookup is performed. • In the event of an L2 cache hit, the data is delivered to the RS and the micro-op's ROB entry in L6 and the micro-op proceeds to the RR stage (see RR entry in this table). • If there is a miss on the L2 cache, the load is forwarded to the bus interface unit for an access to main memory and the micro-op stalls until it has fulfillment. Because the L1 and L2 caches are non-blocking, however, subsequent loads that hit on L1 or L2 can complete before the stalled load.
RR	Ready for retirement. When it has been established that there is no chance of a branch occurring earlier in the program flow, the load micro-op is marked RR in the ROB.
RT	Retirement stage. Data returned by the load is copied to the real target register from the ROB entry and the ROB entry becomes available for a new micro-op.

Handling Stores

An IA store (i.e., memory data write) instruction decodes into two micro-ops. When executed,

- one generates the address to be stored to.
- the other generates the data to be stored .

They can both be dispatched and executed simultaneously because the processor implements two separate execution units for handling stores:

- Store address unit generates the address to be stored to.
- Store data unit generates the data to be written.

Relative to other stores, the processor always performs stores in strict program order. A store cannot be executed until all earlier program stores have been performed. This is necessary in order to ensure proper operation of memory devices that are sensitive to the order in which information is delivered to them. In addition, stores are never speculatively executed. All upstream conditional branches must have been resolved before stores are executed.

When executed, all stores are handled in the following manner:

- If the store is within UC memory, it is posted in the posted-write buffer and, when the posted writes are performed on the bus, will be performed in strict program order relative to other stores. For a description of UC memory, refer to "Uncacheable (UC) Memory" on page 124.
- If the store is within WC memory, it is posted in a WC (write-combining) buffer to be written to memory later. When the WC buffers are written to memory, the write may not occur in the order specified by the program relative to other writes. For a description of WC memory, refer to "Write-Combining (WC) Memory" on page 124.
- If the store is within WT memory and it's a hit on the data or L2 caches, the line is updated and the write is also posted in the posted-write buffer. If the store is a cache miss, it has no effect on the caches, but is posted in the posted-write buffer. When the posted writes are performed on the bus, it will be performed in strict program order relative to other stores. For a description of WT memory, refer to "Write-Through (WT) Memory" on page 126.
- If the store is in WP memory, it has no effect on the caches, but is posted in the posted-write buffer. When the posted writes are performed on the bus, it will be performed in strict program order relative to other stores. For a description of WP memory, refer to "Write-Protect (WP) Memory" on page 126.
- If the store is in WB memory and it's a hit on the data or L2 cache, the write is absorbed into the line and is not posted in either the posted-write or WC buffers. For a description of operation within WB memory, refer to "Relationship of L2 and L1 Caches" on page 147 and to "Write-Back (WB) Memory" on page 127.

The Intel documentation does not specify the depth of the posted-write buffer.

Description of Branch Pr diction

486 Branch Handling

The 486 processor does not implement any form of branch prediction. The instruction decode logic does not differentiate branches from other instruction types and therefore does not alter prefetching. In other words, instruction prefetching always continues past a branch to the next sequential instruction. If the branch is taken when it is executed, the instructions in the prefetch queue and those in the D1 and D2 stages are flushed. This creates a relatively minor bubble in the instruction pipeline (minor because the pipeline has so few stages).

Pentium Branch Prediction

The Pentium processor implements dynamic branch prediction using a very simple mechanism that yields a typical 85% hit rate. As each prefetched instruction is passed into the dual instruction pipelines, the memory address it was fetched from is used to perform a lookup in the BTB (branch target buffer). The BTB is implemented as a high-speed, look-aside cache. If there's a branch and it misses the BTB, it is predicted as not taken and the prefetch path is not altered. If it hits in the BTB, the state of the BTB entry's history bits is used to determine whether the branch should be predicted as taken or not taken. When the branch is executed, its results (whether it was taken or not and, if taken, the branch target address) are used to update the BTB. If the branch is incorrectly predicted, the instructions in the two pipelines and those in the currently-active prefetch queue must be flushed. This doesn't cause a terrible performance hit because the Pentium has relatively shallow instruction pipelines. For a more detailed discussion of the Pentium's branch prediction algorithm, refer to the MindShare book entitled *Pentium Processor System Architecture* (published by Addison-Wesley).

Pentium Pro Branch Prediction

Mispredicted Branches are VERY Costly!

The Pentium Pro processor's instruction pipeline is deep (i.e., it consists of many stages). Close to the beginning of the pipeline (in the IFU2 stage; see Fig-

ure 5-19 on page 111), the address of a branch instruction is submitted to the dynamic branch prediction logic for a lookup and, if a hit, a prediction is made on whether or not the branch will be taken when the branch instruction is finally executed. The BTB only makes predictions on branches that it has seen taken previously. Based on this prediction, the branch prediction logic takes one of two actions:

- If the branch is predicted taken, the instructions that were fetched from memory locations along the fall-through path of execution are flushed from the 16-byte block of code that is currently in the IFU2 stage. The instructions currently in the prefetch streaming buffer are also flushed. The branch prediction logic provides the branch target address to the IFU1 stage and the prefetcher begins to refill the streaming buffer with instructions from the predicted path.
- If the branch is predicted as not taken, the branch prediction logic does not flush the instructions that come after the branch in the 16-byte code block currently in the IFU2 stage. It also does not flush the streaming buffer and the prefetcher continues fetching code along the fall-through path.

The branch instruction migrates through the pre-ROB pipeline stages, is placed in the ROB and, ultimately, is executed by the JEU (jump execution unit). The series of instructions from the predicted path followed along behind it in the pipeline and were also placed in the ROB. Due to the processor's out-of-order execution engine, by the time the branch is executed a number of the instructions that come after the branch in the program flow may have already completed execution.

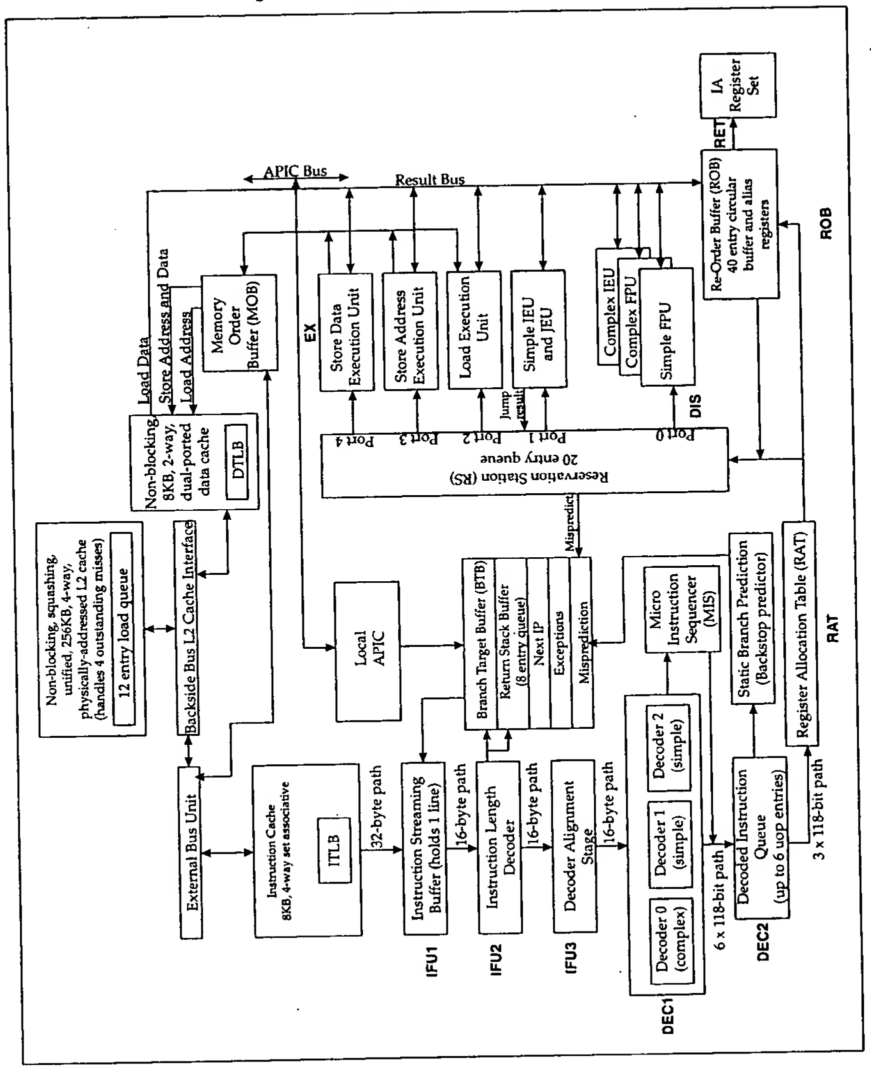
When the branch is finally executed, all is well if the prediction was correct. However, if the prediction is incorrect:

- all instructions currently in the ROB that came after the branch must be flushed from the ROB (along with their execution results if any of them had already been executed).
- all of the instructions that come after the branch that are currently in the RS or are currently being executed must be flushed
- all instructions in the earlier pipeline stages must be flushed
- the streaming buffer must be flushed.
- the prefetcher must then issue a request for code starting at the correct address. This is supplied by the BTB which always keeps a record of the branch target address as well as the fall through address.

Because of the large performance hit that results from a mispredicted branch, the designers of the processor incorporate two forms of branch prediction:

dynamic and static branch prediction, and implemented a more robust dynamic branch prediction algorithm than that used in the Pentium processor. The sections that follow describe the static and dynamic branch prediction mechanisms.

Figure 5-19: Fetch/Decode/Execute Engine


Non-blocking, squashing, unified, 256KB, 4-way, physically-addressed L2 cache (handles 4 outstanding misses)
12 entry load queue
Backside Bus L2 Cache Interface
External Bus Unit
Non-blocking, 8KB, 2-way, dual-ported data cache
DTLB
Load Data
Store Address and Data
Load Address
Memory Order Buffer (MOB)
APIC Bus
Result Bus
Instruction Cache 8KB, 4-way set associative
ITLB
32-byte path
Local APIC
EX
Store Data Execution Unit
Store Address Execution Unit
Load Execution Unit
Simple IEU and JEU
Complex IEU
Complex FPU
Simple FPU
Port 4
Port 3
Port 2
Port 1
Port 0
Jump result
Reservation Station (RS) 20 entry queue
IFU1
Instruction Streaming Buffer (holds 1 line)
16-byte path
IFU2
Instruction Length Decoder
16-byte path
IFU3
Decoder Alignment Stage
16-byte path
Branch Target Buffer (BTB)
Return Stack Buffer (8 entry queue)
Next IP
Exceptions
Misprediction
Mispredict
DEC1
Decoder 0 (complex)
Decoder 1 (simple)
Decoder 2 (simple)
Micro Instruction Sequencer (MIS)
DIS
6 x 118-bit path
DEC2
Decoded Instruction Queue (up to 6 uop entries)
Static Branch Prediction (Backstop predictor)
3 x 118-bit path
Register Allocation Table (RAT)
RAT
Re-Order Buffer (ROB) 40 entry circular buffer and alias registers
RET
IA Register Set
ROB

Dynamic Branch Prediction

General. Refer to Figure 5-19 on page 111. The Pentium Pro processor's BTB implements a two-level, adaptive dynamic branch prediction algorithm. *Please note that the following initial discussion temporarily ignores the static branch prediction mechanism.* In the IFU2 pipeline stage, the address of any conditional branch instruction (up to four simultaneously) that resides within the 16-byte code block is submitted to the BTB for a lookup (in other words, the BTB can make up to four branch predictions simultaneously). The first time that a branch is seen by the BTB, it results in a BTB miss (the BTB has no history on how it will execute). When the branch finally arrives at the JEU (jump execution unit) and is executed, an entry is made in the BTB. The entry records:

- the address that the branch was fetched from.
- whether or not the branch was taken and, if so, the branch target address is recorded in the BTB entry.

The size and organization of the BTB is processor design-specific. The current implementations have a 4-way set-associate BTB with a total of 512 entries (double the Pentium BTB's size).

Yeh's Prediction Algorithm. Unlike the Pentium's BTB, which uses a simple counting mechanism (a variant on the Smith algorithm, a 2-bit counter that increments each time the branch is taken and decrements each time it's not; either of the higher two values indicate a taken prediction, while the lower two values indicate a not taken prediction), the Pentium Pro processor's BTB is capable of recognizing behavior patterns such as taken, taken, not taken. The algorithm used is referred to as Yeh's algorithm and is a two-level, adaptive algorithm. Each BTB entry uses 4-bits to maintain history on the branch's behavior the last four times that it was executed. There are many ways in which this algorithm can be implemented, but Intel has declined to describe the Pentium Pro's implementation. For additional information on Yeh's algorithm, refer to a very good article in the 3/27/95 issue of Microprocessor Report (Volume 9, Number 4). It is estimated that this algorithm can achieve accuracy of approximately 90-95% on SPECint92 and better on SPECfp92.

Return Stack Buffer (RSB)

The programmer calls a subroutine by executing a call instruction, explicitly citing the address to jump to. In other words, a call instruction is an unconditional branch and is therefore always correctly predicted as taken. When the call is executed, the address of the instruction that immediately follows the call is automatically pushed onto the stack by the processor. The programmer always

ends the called routine with a return instruction. The return instruction pops the previously-pushed address off the stack and jumps to the instruction it points to in order to resume execution of the program that called the routine. In other words, the return instruction is also an unconditional branch and can always be predicted as taken. While this is true, it doesn't address the question of where it will branch to.

Previous x86 processor implementations kept no record of the address pushed onto the stack and therefore, although it could be predicted that the branch is taken, it could not "remember" where the branch would be taken to.

Each time that a return instruction is seen, the Pentium Pro processor performs a lookup in a small cache of return instructions (the return stack buffer, or RSB) that it has previously seen executed. The first time the instruction is seen, it results in a miss on the RSB. When the return instruction is executed and the return address is popped from the stack, it is recorded in the RSB for future lookups. The next time that the instruction is seen in the IFU2 stage, it results in a hit on the RSB and the RSB supplies the branch target address to the prefetcher. In cases where the called routine does not alter the pointer that was pushed onto the stack by the call instruction, the return will always be predicted correctly. However, if a routine changes the pointer dynamically, the RSB won't be correct. The Intel documentation is unclear as to the size of the RSB, but the author has seen sizes of 4 and 8 entries quoted.

Static Branch Prediction

The static branch predictor is also referred to as the backstop mechanism because it provides a backup to the dynamic branch prediction logic. Branches are submitted to the static branch prediction logic in the DEC2 stage (see Figure 5-21 on page 115). The static branch prediction decision tree is illustrated in Figure 5-20 on page 114.

For the most part, the static branch predictor handles branches that miss the BTB. However, in the case of an unconditional IP-relative branch, the static branch predictor always forces the prediction to the taken state (and updates the BTB if there was a disagreement).

Figure 5-20: Static Branch Prediction Algorithm

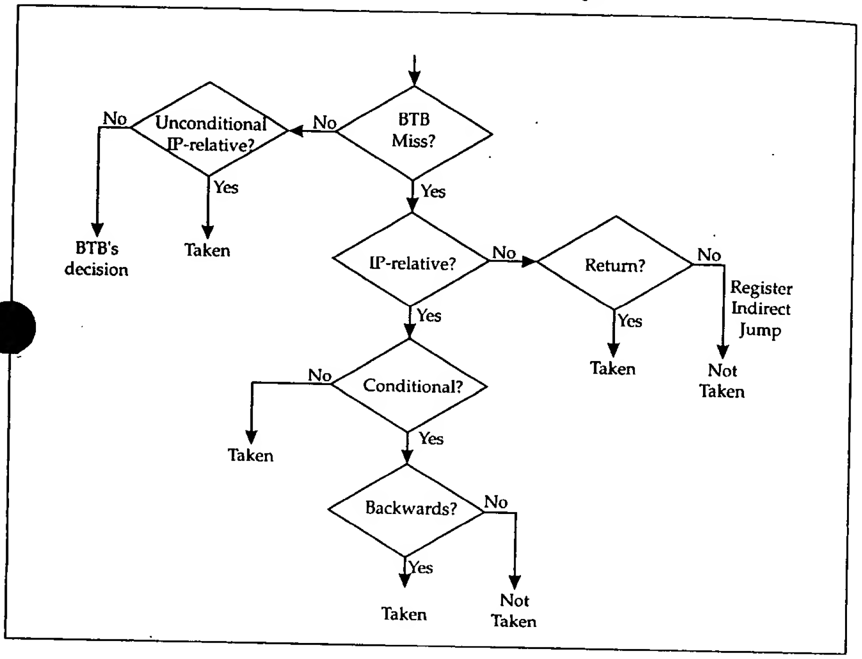
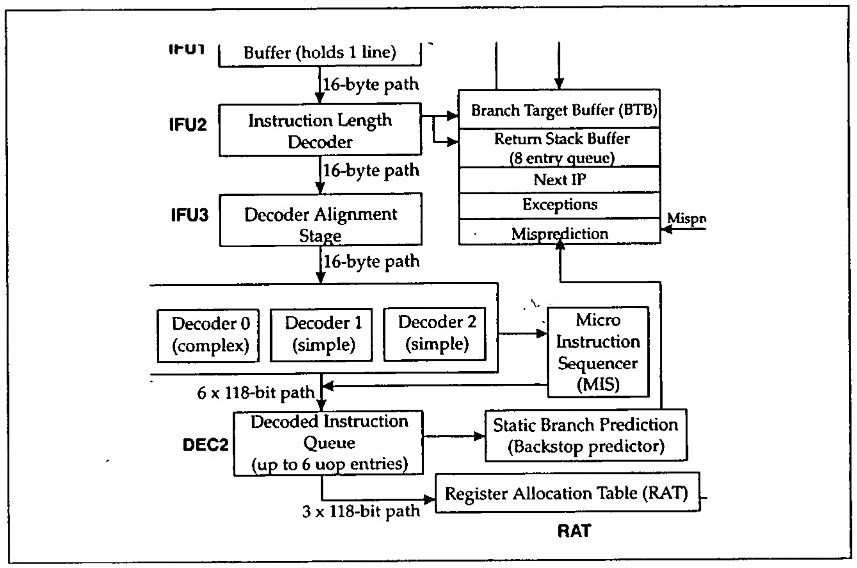


Figure 5-21: Static Branch Prediction Logic



IFU1
Buffer (holds 1 line)
16-byte path
IFU2
Instruction Length Decoder
16-byte path
IFU3
Decoder Alignment Stage
16-byte path
Branch Target Buffer (BTB)
Return Stack Buffer (8 entry queue)
Next IP
Exceptions
Misprediction
Mispr
Decoder 0 (complex)
Decoder 1 (simple)
Decoder 2 (simple)
Micro Instruction Sequencer (MIS)
6 x 118-bit path
DEC2
Decoded Instruction Queue (up to 6 uop entries)
Static Branch Prediction (Backstop predictor)
3 x 118-bit path
Register Allocation Table (RAT)
RAT

Code Optimization

General

This section highlights some of the code optimizations recommended by Intel to yield the best processor performance.

Reduce Number of Branches

As described in "Pentium Pro Branch Prediction" on page 108, mispredicted branches cause a severe performance penalty. Wherever possible, eliminate branches. One excellent way to do this is by using the CMOV and FCMOV instructions (see "Conditional Move (CMOV) Eliminates Branches" on page 367 and "Conditional FP Move (FCMOV) Eliminates Branches" on page 367).

Follow Static Branch Prediction Algorithm

In order to optimize performance for branches that miss the BTB, craft branches to take advantage of the static branch prediction mechanism (see "Static Branch Prediction" on page 113).

Identify and Improve Unpredictable Branches

Indirect branches, such as switch statements, computed GOTOs, or calls through pointers can branch to an arbitrary number of target addresses. When the branch goes to a specific target address a large amount of the time, the dynamic branch predictor will handle it quite well. However, if the target address is fairly random, the branch prediction may not have a very good rate of success. It would be better to replace the switch, etc., with a set of conditional branches that the branch prediction logic can handle well.

Don't Intermingle Code and Data

Don't intermingle code and data within the same cache line. The line can then end up residing in both the code and data caches. If the data is modified, the processor treats this as if it is self-modifying code (see "Self-Modifying Code and Self-Snooping" on page 173), resulting in poor performance.

Align Data

Data objects should be placed in memory aligned within a dword wherever possible. A data object (consisting of 2 or 4 bytes) that straddles a dword boundary may end up causing a lengthy delay to read or write the object. The worst-case scenario would be a data object that straddles a page (i.e., a 4KB) address boundary and causes two page fault exceptions (because neither page is currently in memory). This would result in a stall of the current program until both pages had been read from mass storage into memory.

Avoid Serializing Instructions

Refer to "CPUID is a Serializing Instruction" on page 366. Serializing instructions such as CPUID restrain the processor from performing speculative code execution and has a serious impact on program execution. Sometimes they must be used to achieve a specific goal, but they should be used as sparingly as

possible. The following instructions cause execution serialization:

- Privileged instructions—MOV to control register, MOV to debug register, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, and LTR.
- Non-privileged instructions—CPUID, IRET, and RSM.

Where Possible, Do Context Switches in Software

When the processor hardware is used to perform a task switch (by jumping through a task gate descriptor), the bulk of the processor's register set is automatically saved to the TSS (task state segment) for the program being suspended, and the register set is then reloaded with the register image from the TSS associated with the program being started (or resumed). This save and reload takes quite a bit of time. It may be possible to suspend the current task by saving a smaller subset of the register set (rather than the entire register set), and it may be possible to start (or resume) the new task by loading just a few of the registers. In this case, performance would be aided if the programmer performs the state save and reload using software.

Eliminate Partial Stalls: Small Write Followed by Full-Register Read

If a full register (EAX, EBX, ECX, or EDX) is read (e.g.— mov ebx, eax) after part of the register is written to (e.g.— mov al, 5), the processor experiences a stall of at least 7 clocks (and maybe much longer). The micro-op that reads from the full register is stalled until the partial write is retired, writing the data to the subset of the real IA register. Only then is the value read from the full, real IA register. In addition, none of the micro-ops that follow the full register read micro-op will be executed until the full register read completes.

Since 16-bit code performs partial register writes a lot, the processor suffers poor performance when executing 16-bit code.

Data Segment Register Changes Serialize Execution

16-bit code frequently changes the contents of data segment registers (DS, ES, FS, and GS). Unlike the processor's general-purpose registers, the data segment registers are currently not aliased. A write to the register immediately changes the value in the real, IA data segment register. Micro-ops that reside downstream from the micro-op that changes the register may perform accesses (loads or stores) within the affected data segment. If the processor were permitted to speculatively execute instructions beyond the one that changes the data segment register before that micro-op had completed, they would be using the old, stale contents of the segment register and would address the wrong location.

For this reason, the processor will not execute any instructions beyond the segment register load until the load has completed. The processor's performance degrades because it is restrained from out-of-order execution. *Since 16-bit code changes the contents of the data segment registers a lot, the processor suffers poor performance when executing 16-bit code.*

More than likely, Intel will fix this problem in subsequent versions of the Pentium Pro processor by aliasing the data segment registers as is already done for the general-purpose registers.